UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA TÉCNICA DE TELECOMUNICACIONES

PROYECTO FIN DE CARRERA

ESTUDIO CINEMÁTICO DEL CUERPO HUMANO MEDIANTE KINECT

DANIEL RAMOS GUTIÉRREZ

CURSO 2012-2013

E.U.I.T. TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA
PLAN 2000

TEMA: Estudio cinemático sobre los movimientos del cuerpo humano mediante Kinect

TÍTULO: Estudio cinemático de movimiento del cuerpo humano con KINECT

AUTOR: Daniel Ramos Gutiérrez

TUTOR: Lino García Morales **Vº Bº.**

DEPARTAMENTO: DIAC

Miembros del Tribunal Calificador:

PRESIDENTE: Manuel Vázquez Rodríguez

VOCAL: Lino García Morales

VOCAL SECRETARIO: Antonio Mínguez Olivares

DIRECTOR:

Fecha de lectura: 18 de Septiembre de 2013

Calificación: **El Secretario,**

RESUMEN DEL PROYECTO:

La cámara Kinect, desarrollada por Prime Sense para la consola XBox de Microsoft, ofrece imágenes en RGB de profundidad (gracias a un sensor infrarrojo).

El objetivo de este proyecto es, partiendo de estos datos, la obtención de variables cinemáticas tales como posición, velocidad y aceleración de determinados puntos de control del cuerpo de un individuo (como pueden ser el cabeza, cuello, hombros, codos, muñecas, cadeas, rodillas y tobillos) a partir de los cuales poder extraer patrones de movimiento.

Para ello se necesita un middleware mediante el entorno de libre distribución (GNU) multiplataforma Processing y el contenedor SimpleOpenNI.

Esto ofrece la posibilidad de prescindir del SDK de Microsoft y aplicarlo a varios sistemas operativos.

El resultado del proyecto es útil en aplicaciones para poblaciones con riesgo de exclusión (como es el espectro autista), en telediagnostico, y en general entornos donde se necesite estudiar hábitos y comportamientos a partir del movimiento humano.

Resumen

La cámara Kinect está desarrollada por Prime Sense en colaboración con Microsoft para la consola XBox, ofrece imágenes de profundidad gracias a un sensor infrarrojo. Este dispositivo también incluye una cámara RGB que ofrece imágenes a color además de una serie de micrófonos colocados de tal manera que son capaces de saber de qué ángulo proviene el sonido. En un principio Kinect se creó para el ocio domestico pero su bajo precio (en comparación con otras cámaras de iguales características) y la aceptación por parte de desarrolladores han explotado sus posibilidades.

El objetivo de este proyecto es, partiendo de estos datos, la obtención de variables cinemáticas tales como posición, velocidad y aceleración de determinados puntos de control del cuerpo de un individuo como pueden ser el cabeza, cuello, hombros, codos, muñecas, caderas, rodillas y tobillos a partir de los cuales poder extraer patrones de movimiento. Para ello se necesita un middleware mediante el entorno de libre distribución (GNU) multiplataforma. Como IDE se ha utilizado Processing, un entorno *open source* creado para proyectos de diseño. Además se ha utilizado el contenedor SimpleOpenNI, desarrollado por estudiantes e investigadores que trabajan con Kinect. Esto ofrece la posibilidad de prescindir del SDK de Microsoft, el cual es propietario y obliga a utilizar su sistema operativo, Windows. Usando estas herramientas se consigue una solución viable para varios sistemas operativos. Se utilizado métodos y facilidades que ofrece el lenguaje orientado a objetos Java (Proccesing hereda de este), y se ha planteado una solución basada en un modelo cliente servidor que dota de escalabilidad al proyecto.

El resultado del proyecto es útil en aplicaciones para poblaciones con riesgo de exclusión (como es el espectro autista), en telediagnóstico, y en general entornos donde se necesite estudiar hábitos y comportamientos a partir del movimiento humano. Con este proyecto se busca tener una continuidad mediante otras aplicaciones que analicen los datos ofrecidos.

Abstract

The Kinect camera is developed by PrimeSense in collaboration with Microsoft for the xBox console provides depth images thanks to an infrared sensor. This device also includes an RGB camera that provides color images in addition to a number of microphones placed such that they are able to know what angle the sound comes. Kinect initially created for domestic leisure but its low prices (compared to other cameras with the same characteristics) and acceptance by developers have exploited its possibilities.

The objective of this project is based on this data to obtain kinematic variables such as position, velocity and acceleration of certain control points of the body of an individual from which to extract movement patterns. These points can be the head, neck, shoulders, elbows, wrists, hips, knees and ankles. This requires a middleware using freely distributed environment (GNU) platform. Processing has been used as a development environment, and open source environment created for design projects. Besides the container SimpleOpenNi has been used, it developed by students and researchers working with Kinect. This offers the possibility to dispense with the Microsoft SDK which owns and agrees to use its operating system, Windows. Using these tools will get a viable solution for multiple operating systems. We used methods and facilities of the Java object-oriented language (Processing inherits from this) and has proposed a solution based on a client-server model which provides scalability to the project.

The result of the project is useful in applications to populations at risk of exclusion (such as autistic spectrum), in remote diagnostic, and in general environments that need study habits and behaviors from human motion. This project aims to have continuity using other applications to analyze the data provided.

Resumen

En este documento se presentan el diseño e implementación de un programa software en un entorno de libre distribución (GNU) que extrae un modelo cinemático de una persona en movimiento mediante el dispositivo Kinect. El programa desarrollado permite tanto almacenar dicha información como enviarla a un cliente mediante una arquitectura cliente servidor.

Abstract

This project presents the design and implementation of software over a GNU environment that extracts a human movement cinematic model using Kinect. This program permits store the information and sends it to client by client-server architecture.

A mis padres.

Gracias por darme la oportunidad que no pudisteis tener.

AGRADECIMIENTOS

Este proyecto culmina un duro camino que no hubiera sido capaz de realizar sin el ánimo y apoyo de mis amigos. Especialmente quiero agradecer a Patricia y Javier su generosidad y paciencia conmigo. De igual forma agradecer la ayuda a mi tutor.

INIDICE DE CONTENIDOS

INDICE TABLAS, FIGURAS E IMÁGENES

INDICE DIAGRAMAS Y CÓDIGOS

Capítulo 1. Introducción

"La 2ª Guerra Mundial y la balística nos dieron la computación digital.
La guerra Fría nos dio Internet.
El terrorismo y la necesidad de vigilancia: Kinect".

(M.Webb & B.London, 2010).

La comparación de Kinect con el ordenador personal o Internet puede parecer exagerada pero hay que tener en cuenta que estos dos inventos se desarrollaron con fines militares o gubernamentales, que poco tiene que ver con los usos a los que se les da actualmente. El paulatino crecimiento de proyectos e investigaciones alrededor del PC e Internet han conseguido que estas dos tecnologías sean clave en la vida actual. El desarrollo de Kinect abrió un mundo de posibilidades.

El lanzamiento de Kinect anunciaba una revolución tecnológica comprable a los avances más revolucionarios del siglo XX. Al igual que el PC o Internet el anuncio de Kinect fue la confirmación de que la tecnología con capital militar o estatal caía en manos del público en general.

El reconocimiento facial, el análisis de los movimientos del individuo, la obtención del esqueleto de la persona, la profundidad de imágenes, son tecnologías desarrolladas para detectar amenazas terroristas en espacios públicos que ahora se pueden utilizar para fines civiles: Interfaces software gestuales, scanner 3D, captura de movimiento para animación de personajes 3D, uso en proyectos biométricos, tecnologías para la asistencia de personas con discapacidad, etc.

Aunque esta evolución pueda parecer diversa se puede resumir en una frase. "Por primera vez los ordenadores podrán ver" (Borenstein, 2012). Los resultados del trabajo sobre imágenes fijas y video mediante la combinación de colores (RGB) pierden la mayor parte de las capacidades que proporciona la visión humana: visión espacial, diferenciación de objetos, seguimiento de personas/objetos durante un tiempo y espacio, reconocimiento del lenguaje gestual, etc. Por primera vez con este tipo de cámaras y el procesamiento de imágenes adecuado se podrán crear aplicaciones informáticas que tengan las mismas capacidades que el ojo humano.

1.1. Objetivos

El objetivo principal de este proyecto es la obtención de variables cinemáticas tales como posición, velocidad, y aceleración de determinados puntos de control del cuerpo de un individuo tales como cabeza, cuello, hombros, codos, manos, tronco, tobillos a partir de los cuales poder extraer patrones de movimiento. Para ello se ha desarrollado un software basado en un entorno de libre distribución (GNU) multiplataforma y el contenedor SimpleNI. Ofreciendo la posibilidad de prescindir del SDK de Microsoft y aplicarlo a varios sistemas operativos.

Objetivos secundarios son: conseguir procesar simultáneamente las medidas de los puntos de control del individuo mediante hilos de ejecución; desarrollar el software como una estructura cliente servidor que permita el envío de los datos conseguidos para su estudio.

El resultado del proyecto es de utilidad en aplicaciones para poblaciones con riesgo de exclusión, en telediagnóstico y en general donde se necesite estudiar hábitos y compartimientos a partir del movimiento humano.

1.2. Estado del arte

La tecnología militar está centrando sus esfuerzos en prever y vigilar actos de terrorismo dentro de los propios estados. El aumento de estos actos en los últimos años ha creado la necesidad de aumentar los sistemas de reconocimiento tanto de personas como de objetos sospechosos.

1.2.1. Visión artificial

La visión artificial o visión técnica es un subcampo de la inteligencia artificial cuyo propósito es el de programar software que entienda una escena. Permite la obtención, procesamiento y análisis de cualquier tipo de información especial obtenida a través de imágenes digitales. Los objetivos incluyen, detención, localización y reconocimiento de objetos (por ejemplo caras humanas); seguimiento de un objeto en diversas escenas; mapeo de una escena para generar modelos tridimensionales; estimación de posturas de humanos, etc. Se diferencian las siguientes fases: captación o

obtención de la imagen visual del objeto; instrucciones que son el conjunto de operaciones a realizar; procesado o tratamiento de la imagen mediante las instrucciones y actuación sobre el objeto en función del resultado obtenido.

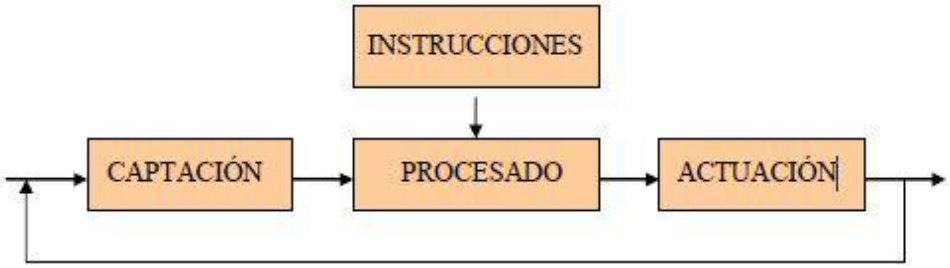

Figura 1. Proceso operativo de la visión artificial

Para la captación de imágenes se utiliza el método digital. La imagen se transforma en valores positivos y enteros que dependen del brillo que presentan los puntos de la imagen original. Estos valores se cuantifican en un nivel de gris.

Las cámaras utilizadas en visión artificial requieren una serie de características que permitan el control del disparo para capturar piezas que pasan por delante. Son más sofisticadas que las cámaras convencionales. Existen varios tipos de cámara según su disposición. Cámaras lineales que construyen una imagen línea a línea realizando un barrido del objeto junto con un desplazamiento longitudinal del mismo. Son usadas en inspección de objetos de longitud indeterminada, telas, papel, vidrio, etc. Cámaras matriciales cuyo sensor cubre un área formada por una matriz de pixeles. Estas cámaras pueden ser monocromo o a color, ofreciendo estas últimas más información a pesar de ser un proceso más costoso.

La evolución en los procesadores de los ordenadores ha conseguido que actualmente sea posible tener software que permita procesos en tiempos suficientemente cortos como para que puedan resolver aplicaciones de visión en entornos de tiempo real. La base de este software es la interpretación y el análisis de los pixeles.

Existen numerosos proyectos en una gran cantidad de campos basados en esta técnica. Algunos ejemplos de aplicaciones en las que se encuentran cámaras y software de visión artificial son desde reconocimiento de madurez de alimentos, aplicaciones sobre videojuegos, hasta reconocimiento y prevención de enfermedades infecciosas.

1.2.1.1. Sistemas de reconocimiento facial

El sistema de reconocimiento facial es una aplicación que identifica automáticamente a una persona en una imagen digital. Esto es posible mediante al análisis de las características o patrones faciales de un sujeto que son extraídos de una imagen o un fotograma de una fuente de video.

Imagen 1. Análisis de estados anímicos mediante reconocimiento facial.[1]

El reconocimiento facial se ha convertido en un área de investigación que abarca diversas disciplinas, como el procesado de imágenes, reconocimiento de patrones, visión por ordenador y redes neuronales. Se podría considerar también dentro del campo de reconocimiento de objetos, donde la cara es el objeto tridimensional sujeto a variaciones de iluminación, pose, etc. Este sistema se utiliza principalmente en sistemas de seguridad, aplicaciones de interacción persona-ordenador, en gestión multimedia, y software fotográfico. Se están creando proyectos que usan los sistemas de reconocimiento facial para un uso comercial (publicidad inteligente). Actualmente ya se aplica en objetos de uso común como por ejemplo en los mecanismos de seguridad de algunos smartphone.

1.2.2. Impresoras 3D y Kinect

Existen proyectos que unen los modelos 3D que nos ofrece Kinect con la impresión 3D. Gracias a nuevos proyectos de código abierto *open source* como por

[1] Ilustración obtenida de http://seetio.com/blog/2010/06/22/maquina-expendedora-reconocimiento-facial-recompensa-helados-gente-feliz-video/

ejemplo RepRap (Replicating Rapid Prototyper) se ha conseguido reducir el precio de las impresoras 3D. Esto ha animado a nuevos artistas digitales como el colectivo barcelonés blablaLAB a crear nuevas iniciativas donde mediante Kinect se reconstruye en 3D el modelo de la persona en una cierta postura y luego se fabrica su figura en plástico.

Imagen 2. Figuras creadas por el colectivo blablaLAB[2]

1.2.3. Tratamientos médicos

El uso de los videojuegos en la recuperación de pacientes se ha convertido en una herramienta más. En la actualidad las posibilidades Kinect están siendo aplicadas en el campo de la medicina como terapias alternativas. El Hospital Universitario de Burgos trata a sus pacientes con problemas de movilidad mediante juegos comerciales basados en Kinect. La eficacia de estas terapias es similar a las clásicas, lo que cambia es la actitud de los pacientes. Ahora mediante los videojuegos se encuentran más motivados en procesos largos. Gracias a las consolas los especialistas cambian la clásica tabla de ejercicios por juegos interactivos. Poco a poco estas técnicas están entrando en más hospitales.[3]

---

2 http://blogs.elpais.com/arte-en-la-edad-silicio/2013/01/esculpe-tu-mundo.html

[3] Agudo, A. (Agosto, 2013). *Para su enfermedad tome este videojuego.* El País.

Capitulo 2. Base teórica

2.1. Cinemática

La cinemática es una rama de la física que estudia las leyes del movimiento de los cuerpos sin atender a las causas que lo provocan. Por tanto la cinemática sólo estudia el movimiento en sí, a diferencia de la dinámica que estudia las interacciones que lo producen. Los conceptos con los que trabaja la cinemática son varios: punto que se refiere a un elemento sin volumen situado en el espacio; sistema de referencia es aquel sistema coordenado con respecto al cual se da la posición de los puntos y el tiempo; posición o localización con respecto a un sistema de referencia; y tiempo.

Las magnitudes cinemáticas son desplazamiento o la diferencia de posición entre un punto y su origen (figura 2.1), velocidad que define el cambio de posición de un punto por unidad de tiempo (figura 2.2) y aceleración o cambio de velocidad de un punto por unidad de tiempo (figura 2.2).

$$\|\Delta \mathbf{r}(t)\| \leq L_r = \int_0^t v(t)\, dt = \int_0^t \left\| \frac{d\mathbf{r}(t)}{dt} \right\| dt$$

Figura 2.1. Distancia[4]

$$\bar{\mathbf{v}} = \frac{\Delta \mathbf{r}}{\Delta t} \qquad\qquad < \mathbf{a} >= \bar{\mathbf{a}} = \frac{\Delta \mathbf{v}}{\Delta t}$$

Figuras 2.2. Velocidad **Aceleración**

2.2. Sistema cegesimal

El sistema cegesimal (CGS) ha sido remplazado por el Sistema Internacional (SI) de Unidades, sin embargo aún perdura su utilización en algunos campos científicos y técnicos muy concretos, por ejemplo en electromagnetismo.

En este proyecto se utilizan unidades CGS en el cálculo de distancias (cm), velocidades (cm/s) y aceleraciones (cm/s^2) de los puntos de control.

[4] Formulas extraídas de Goldstein, H. *Mecánica Clásica* 2ªEd. Reverte. 2006.

Magnitud	Definición	Equivalencia SI
Longitud	cm	0,01m
Velocidad	cm/s	0,01m/s
Aceleración	cm/s^2	$0,01m/s^2$

Tabla 1. Equivalencia CGS-SI

2.3. Estudio de Kinect

Kinect es una cámara de profundidad, se basa en un emisor infrarrojos y una cámara. Las cámaras normales producen imágenes en los que cada píxel registra el color de la luz (RGB) que rebota de los objetos. Kinect, por su parte, registra la distancia de los objetos que se encuentren en la escena creando una imagen de profundidad. Para ello se utiliza luz infrarroja que no capta el aspecto de los objetos sin no su posición en la escena. Esta imagen de profundidad se muestra en blanco y negro con algo de distorsión. Las partes más claras son las más cercanas y las más oscuras son las más alejadas. Mediante el uso de Processing (u otros entornos de desarrollo similares) es posible crear programas que nos den la distancia de los usuarios incluso en movimiento.

Imagen 3. Kinect sin su carcasa de plástico, muestra de izquierda a derecha el emisor infrarrojos (IR), la cámara RGB y la cámara IR.[5]

Microsoft (en su división Microsoft Research) desarrollo Kinect como un periférico para la consola de videojuegos Xbox 360. Kinect es el producto de muchos

[5] Imagen obtenida de iFixit http://www.ifixit.com/Teardown/Microsoft+Kinect+Teardown/4066/2

años de investigación académica realizada tanto por Microsoft como por diversas instituciones y empresas, por ejemplo la Universidad de Washington.

El hardware de Kinect fue desarrollado por PrimeSense, una empresa israelí que ha producido anteriormente otras cámaras de profundidad utilizando la misma técnica basada en proyección IR. PrimeSense colaboró con Microsoft para producir una cámara de profundidad que trabajar con el software y los algoritmos que Microsoft había desarrollado en sus investigaciones. PrimeSense es propietaria de la tecnología hardware de la cámara, de hecho, PrimeSense anunció que se está trabajando con ASUS para crear una cámara de profundidad similar al Kinect destinada a la integración con televisores y ordenadores para aplicaciones y juegos.

2.3.1. Funcionamiento interno

En la imagen 3 y 4 vemos las dos cámaras que usa Kinect que a diferencia de las cámaras tradicionales (digitales, web, etc.) tiene además un emisor de infrarrojos. La luz infrarroja tiene una longitud de onda más larga que la de la luz visible y no se puede ver a simple vista.

Además de las cámaras Kinect tiene otros cuatro sensores que la diferencian de una cámara de profundidad corriente: micrófonos. Estos micrófonos se distribuyen en todo el Kinect de una manera muy parecida a los oídos humanos. Su objetivo no es sólo escuchar los sonidos, para ello un micrófono habría sido suficiente. Mediante el uso de varios micrófonos juntos, Kinect no sólo puede capturar el sonido, sino también localizar el sonido dentro de la sala. Por ejemplo, si varios jugadores están usando comandos de voz para controlar un juego Kinect puede saber qué comando está viniendo de cada jugador. Esta es una función potente y muy interesante.

Dentro de la base de plástico de Kinect hay un pequeño motor y una serie de engranajes. Al activar este motor Kinect puede inclinar sus cámaras y altavoces de arriba a abajo. El rango de movimiento del motor está limitado a unos 30 grados. Microsoft añade el motor para mover la cámara y trabajar dependiendo del tamaño de la habitación y la posición de los muebles respecto a las personas que juegan con la Xbox.

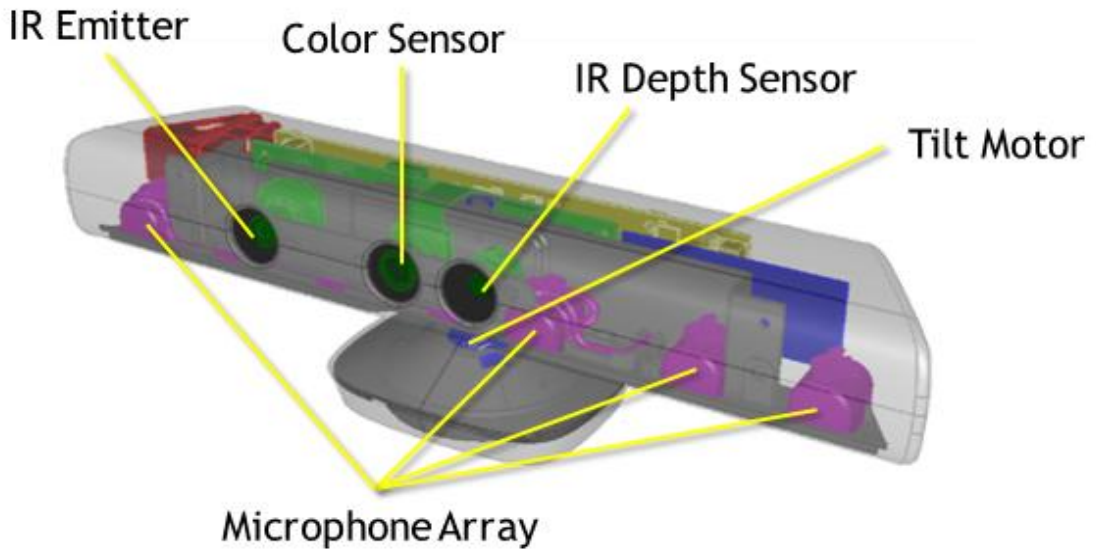

Imagen 4. Vista interna de los dispositivos de Kinect.[6]

Añadir que tanto las características de audio de Kinect como la parte del motor no estarán disponibles en la biblioteca que se va a utilizar. Como información adicional comentar que el kit de desarrollo de Microsoft para Kinect si incluye opciones de audio.

Ni el audio ni la capacidad motriz son objetivo de este proyecto, por tanto no se profundizara más en estas características.

2.3.1.1. Cámara y emisor infrarrojo

El emisor proyecta un conjunto de putos de luz infrarroja sobre la escena. Los puntos son invisibles para el usuario, pero es posible capturar una foto de ellos mediante una cámara IR. La cámara infrarroja es un sensor específicamente diseñado para captar luz infrarroja. En la imagen 3 se puede ver la lente de un color verdoso a diferencia de la cámara normal. Por lo tanto, Kinect puede ver la red de puntos de infrarrojos que se proyecta sobre los objetos que se encuentren en frente.

Kinect se calibra proyectando puntos de luz infrarroja sobre una pared plana a una distancia conocida. De esta manera los puntos dan unos valores de profundidad y se toman como referencia para futuras medidas. Cualquier objeto que está más cerca o más lejos que la distancia de calibración de Kinect va a ser representado con puntos fuera de su posición. Dado que Kinect está calibrado para conocer la posición original

[6] Ilustración obtenida de http://msdn.microsoft.com/en-us/library/jj131033.aspx

de todos los puntos puede utilizar su desplazamiento para calcular la distancia de los objetos. Esto hace que Kinect pueda convertir esta imagen IR de una red de puntos de profundidad.

Existen ciertas limitaciones que son inherentes al funcionamiento de este sistema. Por ejemplo, los objetos que se encuentren detrás de otros no serán expuestos a la luz infrarroja por tanto no se tendrán datos de profundidad de estos, estos objetos están en sombra. Kinect solo conoce los objetos "golpeados" por su emisor IR.

2.3.1.2. Cámara RGB

Kinect también dispone de una cámara de color. Esta cámara tiene un sensor digital que es similar a la que en muchas cámaras web y pequeñas cámaras digitales. Tiene una resolución relativamente baja (640 x 480 píxeles), aunque es posible aumentar esta resolución hasta 1280 x 1024 a costa de bajar el *frame rate* a de 30 a 10 *frames* por segundo (fps).

Imagen 5. Modelo 3D coloreado.[7]

Por sí misma la cámara de color no es particularmente interesante. Es sólo una webcam de baja calidad. Pero Kinect puede alinear la imagen en color a partir de la información de profundidad capturada por su cámara de IR. Eso significa que es posible

[7] Imagen obtenida : Making Things See – Greg Boreestein

alterar el color de la imagen en función de su profundidad (por ejemplo, ocultando nada más que una cierta distancia). Y a la inversa, es posible reproducir el color real en las imágenes 3D creadas a partir de la información de profundidad, creando escaneados 3D o entornos virtuales con color realista.

2.3.2. La imagen de profundidad

En primer lugar, una imagen de profundidad es mucho más fácil de entender para un equipo que una imagen convencional. Los programas de tratamiento de imágenes comienzan analizando los píxeles y tratando de encontrar y reconocer a las personas y objetos que se representan. Un programa informático que trabaja con imágenes a color tiene muy difícil distinguir los objetos y las personas. La mayor parte del color de los píxeles se determina por la luz de la habitación en el momento en que se capturó la imagen, la abertura y otros factores de la cámara.

En una imagen de profundidad, a diferencia de las convencionales, el color de cada píxel indica la distancia a la que la imagen esta de la cámara. Los valores corresponden directamente al lugar donde están los objetos en el espacio, esto es mucho más útil al determinar dónde comienza un objeto y donde otros, y si hay alguna persona alrededor. También, debido a que Kinect crea su imagen de profundidad (se explica más adelante) de forma que no es sensible a las condiciones de luz en la imagen tendrá los mismos valores de distancia tanto en una habitación iluminada como en una completamente oscura. Esto hace que las imágenes de profundidad sean más fiables.

Una imagen de profundidad también contiene información tridimensional precisa. Podemos utilizar los datos de una cámara de profundidad, como Kinect para reconstruir un modelo en 3D de lo que ve la cámara. Podemos entonces manipular este modelo, viéndolo desde ángulos diferentes, combinándolo con otros modelos 3D preexistentes, e incluso utilizarlo como se comenta en el capítulo 1 como parte de un proceso de fabricación digital para producir nuevos objetos físicos.

Gracias a las características de las imágenes de profundidad podemos utilizarlas para detectar y rastrear a las personas incluso la localización de las articulaciones y partes del cuerpo. Microsoft ha desarrollado el Kinect específicamente para explotar esta capacidad de detección y usarla en videojuegos. Afortunadamente, tenemos acceso

a un software que puede realizar este proceso y simplemente nos da la ubicación de los usuarios. Nosotros no tenemos que analizar la imagen de profundidad con el fin de obtener esta información.

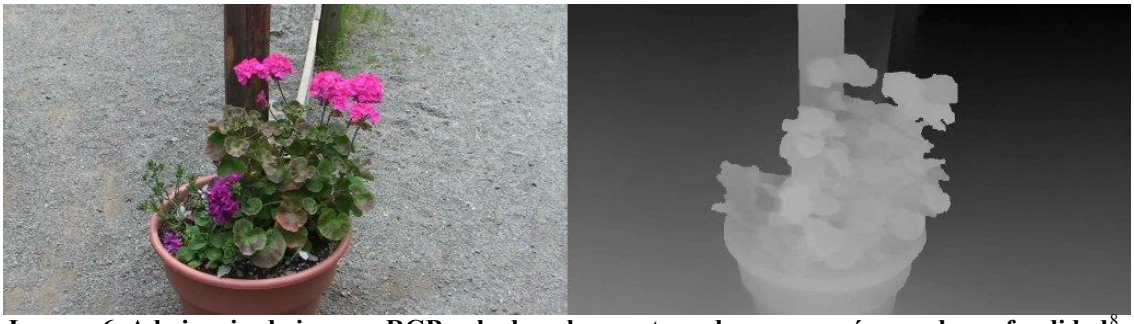

Imagen 6. A la izquierda imagen RGB, a la derecha una tomada por una cámara de profundidad[8].

2.3.2.1. Imágenes y pixeles

Una imagen, si se trata de una imagen de profundidad del Kinect, es una simple colección de píxeles. Un píxel es la unidad más pequeña de una imagen. Cada píxel es de un solo color sólido y mediante la disposición de muchos píxeles próximos el uno al otro las imágenes digitales puede crear ilusiones tales como gradientes suaves y sutil sombreado. Cuanto mayor es la cantidad de píxeles más suave es el gradiente y mejores resultados obtenemos, véase las imágenes digitales.

Dependiendo del tipo de imagen, los colores de un píxel puede varían. En una imagen en blanco y negro, cada píxel es negro, blanco o algún tono de gris. En una imagen en color, cada píxel tiene un componente rojo, verde y azul (RGB), que se combinan en cantidades diferentes para crear cualquier color.

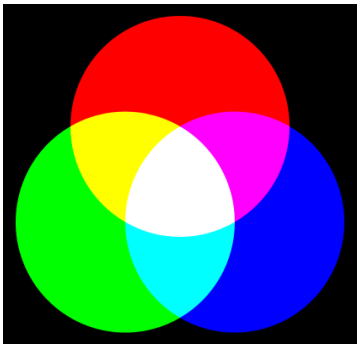

Imagen 7. Modelo aditivo de colores rojo, verde, azul.[9]

[8] Ilustración obtenida de http://www.staff.science.uu.nl/~tan00109/student/2010_e_bob/
[9] Ilustración obtenida de http://docs.gimp.org/

Cada componente o color tiene una gama que va desde la máxima intensidad hasta el negro. Por lo tanto un píxel rojo, por ejemplo, va desde un tono negro pasando por marrón oscuro, hasta un rojo brillante. Cuando el componente rojo se combina con los otros componentes el píxel puede representar cualquier tono en un amplio espectro de colores. El valor numérico de cada pixel representa su intensidad. En una imagen de escala de grises, el valor numérico representa el grado de oscuridad del píxel donde 0 es negro y 255 es blanco. En una imagen en color, cada componente del pixel (rojo, verde, y azul) puede variar de 0 a 255, donde 0 es negro y 255 es el color primario.

La gama de colores que puede representar un píxel está determinada el número más grande que se utilizar para representar cada componente. Cuanto mayor sea el número que estamos dispuestos a aceptar para cada píxel, más memoria va a ocupar el píxel. Y ya que las imágenes pueden tener millones de píxeles, la suma de estos hace que la memoria total de la imagen aumente en gran medida. Las imágenes que utilizan números grandes para almacenar cada componente del píxel ocuparan enormes cantidades de espacio en memoria llegando a ser inmanejables a la hora de procesar. Por lo tanto, muchos entornos de programación han optado por un valor máximo de píxeles más moderado 255, pero todavía lo suficientemente grande como para conseguir imágenes realistas. Este valor se conoce como profundidad de bits de la imagen. Los entornos de programación como Processing almacenan estos píxeles como números de 8 bits (de 0 a 255).

Imagen 8. Uso de diferentes formas de pixeles, de izq. a derecha puntos, líneas y un filtro de suavizado.

Dependiendo de la resolución, cada imagen digital tiene miles o millones de píxeles dispuestos en una cuadrícula que se compone de columnas y filas. Al igual que con cualquier red, el número de columnas es igual a la anchura de la imagen, y el número de filas es igual a su altura. La profundidad de imagen capturada por la Kinect

es de 640 píxeles de ancho por 480 píxeles de alto. Esto significa que cada imagen de profundidad tendrá un total de 307.200 píxeles. Y a diferencia de la escala de grises e imágenes a color, cada píxel va a hacer una doble función. El número de cada píxel, entre 0 y 255, representa tanto un color de gris como una distancia en el espacio. Podemos utilizar el mismo número para cualquiera de las dos funciones, podemos tratar a 0 como negro y 255 como blanco, o podemos tratar a 0 como muy lejano y 255 como muy próximo. En resumen, Kinect muestra es a la vez una imagen que representa la escena y una serie de mediciones de la misma.

2.4. Entornos de desarrollo open source

Como se dijo en el punto 2.3 Kinect fue desarrollado por varias empresas, al principio la combinación de software de Microsoft con el hardware de PrimeSense era conocido por su nombre en clave *Project Natal*. En 2010, el dispositivo se puso en marcha, "Microsoft Kinect" salió a la venta al público por primera vez. Kinect fue un gran éxito comercial. Vendió al alza de 10 millones de unidades[10] en el primer mes de lanzamiento, por lo que es el periférico que ha vendido más unidades en menos tiempo de toda la historia.

Sin embargo, para este proyecto, tal vez un hito aún más importante fue la creación de controladores de código abierto, *open source,* para el Kinect. Tan pronto como el Kinect salió al mercado los programadores de todo el mundo comenzaron a trabajar en la creación de controladores de código abierto que permitirían a cualquier persona acceder a los datos de Kinect.

Kinect se conecta al ordenador a través de USB (véase Apéndice), al igual que muchos otros dispositivos, como ratones, teclados y cámaras web. Todos los dispositivos USB requieren software y drivers para operar. Los controladores son piezas especiales de software que se ejecutan en el ordenador y lo comunican con el hardware externo. Una vez que se tenga el controlador para un determinado hardware, ningún otro programa necesita entender cómo hablar con ese dispositivo. El programa no necesita saber acerca de la marca de la cámara web, ratón, etc. debido a que su equipo tiene un

[10] Reuters (Marzo, 2011) *Microsoft vende 10 millones de Kinect.* http://www.elmundo.es/

controlador que lo hace accesible a todos los programas. Microsoft sólo diseñó Kinect para trabajar con Xbox 360, por lo que no dio a conocer los controladores que permiten que los programas de ordenador tengan acceso a la Kinect. Al crear un controlador de código abierto para Kinect se trata de hacer que el Kinect pueda ser usado para todos los programas en todos los sistemas operativos.

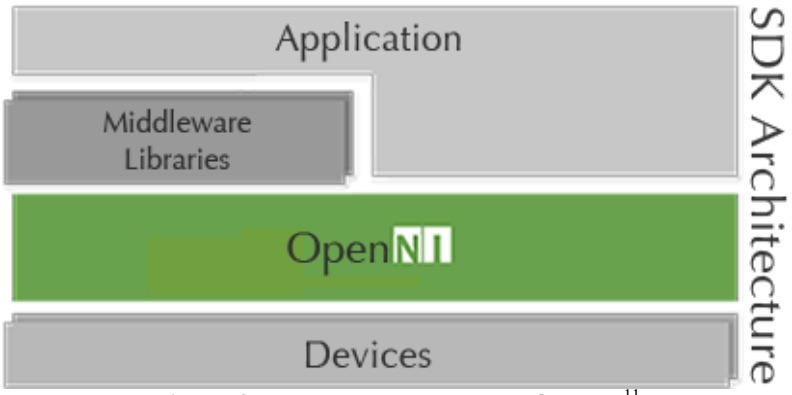

Figura 3. Estructura del entorno OpenNI[11]

Tras el gran interés de desarrolladores y estudiantes en Kinect y la investigación de programas de código abierto como OpenKinect que explotaran sus posibilidades, PrimeSense lanzó su propio software para trabajar con Kinect. Además de drivers que permitieron a los programadores acceder a la información de Kinect. PrimeSense incluye software más sofisticado que procesa la imagen profundidad para detectar usuarios y localizar la posición de sus articulaciones en tres dimensiones. Llamaron a su sistema OpenNI (*Open Natural Interaction*). OpenNI representa un gran avance para los desarrolladores que trabajan con Kinect. Por primera vez los datos de los usuarios que hicieron de Kinect una gran herramienta para la construcción de proyectos interactivos estaban disponibles para proyectos personales.

Los datos de usuario proporcionados por OpenNI dan una información exacta de la posición de las articulaciones de los usuarios (cabeza, hombros, codos, muñecas, pecho, caderas, rodillas, tobillos y pies) en todo momento mientras se está utilizando. Esta información es fundamental para aplicaciones interactivas. Si se desea que los usuarios sean capaces de controlar su aplicación a través de gestos o movimientos, o su posición dentro de la habitación, los mejores datos son las posiciones exactas de las

[11] Figura obtenida de http://www.openni.org/

manos, las caderas y los pies. Si bien el proyecto OpenKinect puede llegar a proporcionar estos datos, y el SDK de Microsoft proporciona los mismos a los desarrolladores que trabajan solamente en Windows, en este momento OpenNI es la mejor opción para quien quiera trabajar en cualquier plataforma y lenguaje de programación.

2.4.1. ¿OpenKinect u OpenNi?

Una ventaja de OpenKinect es su licencia de software. Los colaboradores del proyecto OpenKinect lanzaron sus drivers bajo una licencia de código completamente abierto, en concreto una doble licencia Apache 2.0 2.0/GPL. Sin entrar en detalles, el significado es que se puede utilizar código OpenKinect en sus proyectos comerciales, sin tener que pagar una cuota de licencia o preocuparse por la propiedad intelectual.

La situación con OpenNI, por otro lado, es más compleja. PrimeSense ha proporcionado dos piezas de *software*. En primer lugar el marco OpenNI. Esto incluye los controladores para acceder a los datos de profundidad básicos de Kinect. OpenNI está bajo la LGPL, una licencia similar en espíritu a la licencia de OpenKinect. Sin embargo, una de las características más interesantes de OpenNI es el seguimiento de los usuarios. Esta función no está cubierta por la licencia LGPL OpenNI. En su lugar esta proporcionada por un módulo externo llamado NITE. NITE no está disponible bajo una licencia de código abierto. Se trata de un producto comercial que pertenece a PrimeSense. Su código fuente no está disponible. PrimeSense proporciona una licencia gratuita que puede utilizarse para hacer proyectos que utilizan NITE con OpenNI, los términos no son claros sobre si se puede usar esta licencia para producir proyectos comerciales.

OpenNI se rige por un consorcio de empresas liderado por PrimeSense así como el fabricante de componentes informáticos ASUS. OpenNI está diseñado para trabajar con Kinect además de con otras cámaras de profundidad. Ya es compatible con cámaras de ASUS como la XTion. Esta es una gran ventaja porque significa que los programas que creemos usando OpenNI para trabajar con Kinect seguirán siendo útiles con las futuras cámaras de profundidad, eliminando la necesidad de volver a escribir nuestras aplicaciones en función de las nuevas cámaras que se quieran usar.

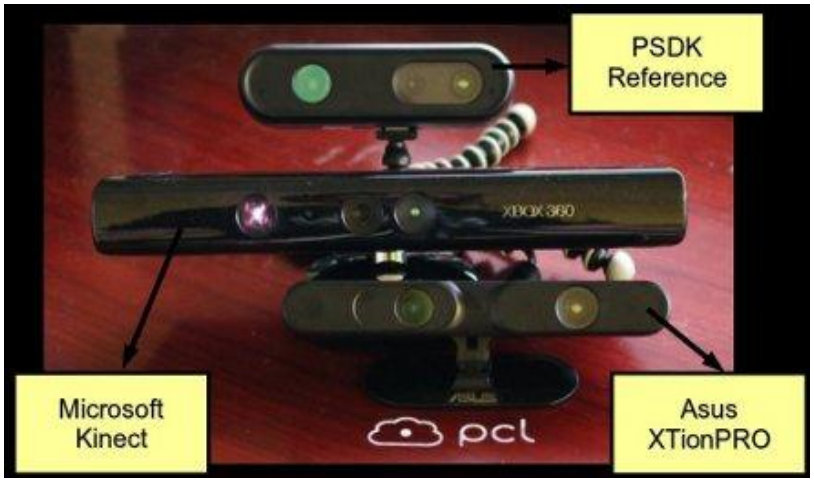

Imagen 9. Diferentes cámaras de profanidad entre ellas Kinect y XTionPro de ASUS[12].

Todos estos factores influyen en la decisión sobre qué plataforma usar. Por un lado, la licencia abierta de OpenKinect y la comunidad que la rodea. Gracias a ella evolucionó el software de código abierto con gran rapidez y entusiasmo. Por otro lado, OpenNI ofrece algunas ventajas técnicas fundamentales, la capacidad de trabajar con los datos de seguimiento de usuarios. Dado que esta característica es el factor clave en este proyecto y la gran diferencia de Kinect respecto las demás cámaras para crear aplicaciones interactivas. En este contexto, la posibilidad de que el código escrito con OpenNI siga funcionando en Kinect, o incluso en una evolución de la misma u otras cámaras de profundidad de próxima generación, es difícil pasar por alto.

Tomando todos aspectos en cuenta, se ha optado por utilizar OpenNI como base del código en este proyecto. Afortunadamente, hay una cosa que tanto OpenNI y OpenKinect tienen en común y es una buena biblioteca de procesamiento. Max Rheiner, artista y profesor en el departamento de diseño de la Universidad de Zúrich, creó una biblioteca de procesamiento que funciona con OpenNI para el uso de sus alumnos. Esta biblioteca se llama SimpleOpenNI y es compatible con muchas de las características más avanzadas de OpenNI. SimpleOpenNI viene con un instalador que hace que sea fácil de instalar, véase el Apéndice A.

[12] Ilustración obtenida de http://docs.pointclouds.org/trunk/group__io.html

2.4.2. Processing

Processing es un lenguaje de programación basado en Java, entorno de desarrollo integrado (IDE) de código abierto y comunidad online. Desde 2001 Processing ha promovido la creación de software dentro de las artes visuales y la creación visual dentro de la tecnología. Inicialmente fue pensado para servir como entorno de desarrollo y para enseñar los fundamentos de programación dentro de un contexto visual, poco a poco ha terminado convertido en una herramienta de desarrollo para los profesionales. Hoy en día, hay decenas de miles de estudiantes, artistas, diseñadores, investigadores y aficionados que utilizan Processing para el aprendizaje, creación de prototipos y producción.[13] Fue iniciado por Ben Fry y Casey Reas en el Instituto Tecnológico de Massachusetts (MIT) contritamente en el MIT Media Lab. Al ser una herramienta basada en Java puede heredar todas su funcionalidades, esto hace que desarrolladores con conocimientos de Java a nivel medio puedan realizar una gran cantidad de proyectos complejos. En este proyecto Processing sirve como medio para trabajar y reproducir las imágenes que nos ofrece Kinect. Si se observa el API de Processing es fácil darse cuenta la gran cantidad de métodos que tiene para trabajar con la imagen o el audio.

Mediante Processing podemos dibujar gráficos en dos y tres dimensiones. El renderizado por defecto es en 2D, si se tiene una tarjeta gráfica compatible con OpenGL podremos tener renderizado en tres dimensiones. Si estamos en un entorno 3D Processing permite controlar la cámara, iluminación y materiales.

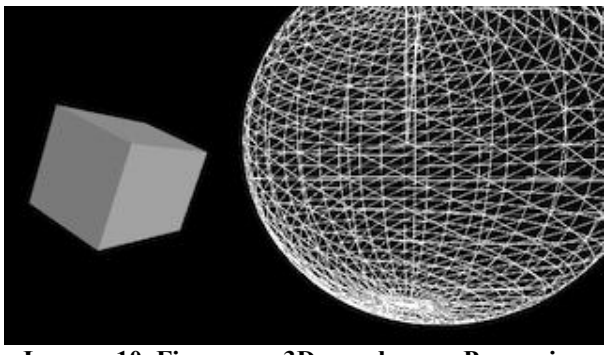

Imagen 10. Figuras en 3D creadas con Processing

[13] Fuente: http://www.processing.org/

Es posible ampliar las capacidades mediante el uso de extensiones y bibliotecas. Las bibliotecas hacen posible extender la programación más allá de los métodos nativos. Hay multitud de bibliotecas creadas por usuarios dentro de la comunidad capaces de aportar nuevas capacidades de sonido, imagen, geometría 3D, comunicación, dispositivos, etc. Cabe destacar la unión de Processing con plataformas como Eclipse y sistemas como Android para crear nuevas aplicaciones en terminales móviles.

2.4.2.1. Entorno de desarrollo

El entorno de desarrollo de Processing (PDE) es sencillo y fácil de usar. Los programas se escriben en el editor de texto y se ejecutan pulsando *start.* Cada programa se escribe en un *sketch* que por defecto es una subclase de *PApplet* de Java que implementa la mayor parte de las funciones de Processing. También permite crear clases propias en el *sketch* pudiendo así usar tipos de datos complejos con argumentos y evita la limitación de utilizar exclusivamente datos tales como enteros, caracteres, números reales y color. Los *sketchs* se almacenan en *sketchbook* que no es más que un directorio dentro del ordenador. Para abrir los bocetos se deberá pulsar *open.*

Imagen 11. Entorno de desarrollo para Mac[14].

[14] Obtenidas de http://www.processing.org/reference/environment/ Version 2.0

La imagen 11 muestra la ventana principal del programa de desarrollo. En la barra de herramientas (*toolbar*) se encuentran los botones básicos. Más abajo (*tabs*) se encuentran las pestañas con los *sketchs* o trozos de código (recordar que por defecto son sub clases de *PApplet* no clases desde cero). El *Text editor* muestra el código correspondiente a cada *sketch.*

Los botones que se encuentra en la barra de tareas son: *play* o inició, ejecuta el programa. *Stop* parar, para el programa y cierra la ventana *display.* Nuevo *sketch,* inicia un nuevo esquema en blanco, *save, load, export,* guardar, cargar y exportar el programa.

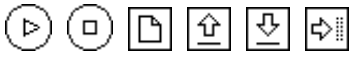

Imagen 12. Botones de la barra de herramientas.

Por último en la parte inferior están las áreas reservadas para la depuración del código. *Message area* donde se muestran los fallos de compilación del programa y *console* similar a una ventana de comandos pero sin la posibilidad de introducir parámetros. En esta ventana *console* podremos ver los fallos en tiempo de ejecución y sacar mensajes, valores de parámetros, etc. que no sirvan como ayuda a la hora de programar. La ventana o *display window* puede ser la parte más distinta si la comparamos Processing con otros entornos de desarrollo como Eclipse o NetBeans. En esta ventana se mostrara la ejecución del programa que hemos escrito. Se podría asemejar a una ventana de la clase grafica *swing* de Java.

2.4.2.2. Métodos básicos de Processing

Para conocer mejor la manera de programar con Processing a la hora de crear aplicaciones es necesario hablar de los métodos principales y las variables fundamentales que usaremos en este proyecto. Recordar que todas ellas heredan del lenguaje de programación orientado a objetos Java, por lo tanto se dan por supuestos los conocimientos básicos de este lenguaje.

Processing se basa en dos métodos fundamentales. El método *setup* y el método *draw.*

- *Método Setup*

Básicamente dentro de este método se encuentra la creación e inicialización de las variables que vayamos a usar. *Setup()* se ejecuta nada más dar al *play* de nuestro interfaz. Es normal encontrar en él funciones como *size()* o *frameRate()* que nos permiten modificar las opciones de la ventana de trabajo. En nuestro caso en este punto lanzaremos el hilo servidor.

- *Método draw.*

El método *draw()* es la parte fundamental de Processing, actuara como el hilo principal de cualquier programa, en el se encuentra la lógica de la aplicación y se llamara a distintas clases si es necesario. Gracia a él podemos plasmar o dibujar lo que queremos en la pantalla. Este método actuara como un bucle infinito y su frecuencia de repetición depende de la tasa de refresco o *frame rate* que queramos (en nuestro caso 30 fps).

También cabe destacar los métodos de entrada desde teclado o ratón. Estos harán que nuestra aplicación pueda ser controlada por el usuario. Son realmente sencillos de utilizar, se basan en eventos. Se explicara *keyPressed()* ya que es el que se utiliza en este proyecto.

Sencillamente este método es invocado cuando se pulsa una tecla durante la ejecución de la aplicación. Como se puede ver en el código siguiente (código 2.1) gracias a los métodos *if* según la tecla pulsada el programa se comportara de una forma distinta.

```
void keyPressed() {
  if (key == 'r') {
    startTime = clock.startTime();
    loop();
  }if (key == 'f') {
    noLoop();
  }if (key == ESC) {
    servidor.pararHilo();
    pararHilos();
  }
}
```

Código 2.1 Métodos de entrada por teclado de Processing.

Existen multitud de variables en Processing. Tipos de datos primitivos como *int*, *String*, *float*… y otros más complejos necesarios para crear figuras geométricas. En nuestro caso el más interesante es *PVector.* Básicamente es una variable que almacena hasta tres valores de tipo *float*, en x, y, z. Esencial para trabajar con puntos en el espacio como es nuestro caso. Más adelante (capítulo 4) se tratara más sobre las variables particulares de Processing.

2.5. Conceptos básicos sobre hilos en Java

La programación multihilo en Java se basa en el concepto de hilo. Un hilo es un único flujo de ejecución dentro de un proceso, entendiendo como proceso el programa que se ejecuta dentro de su propio espacio de direcciones. Java permite esto ya que es un sistema multiproceso, es decir soporta varios procesos a la vez corriendo de forma simultánea. Un concepto más familiar es el de multitarea.

Retomando la idea de hilo diremos que es una secuencia de código en ejecución dentro del contexto de un proceso. Los hilos no pueden ejecutarse de forma independiente, requieren de la supervisión de un proceso padre. Esto significa que los hilos están siempre asociados a un proceso en particular.

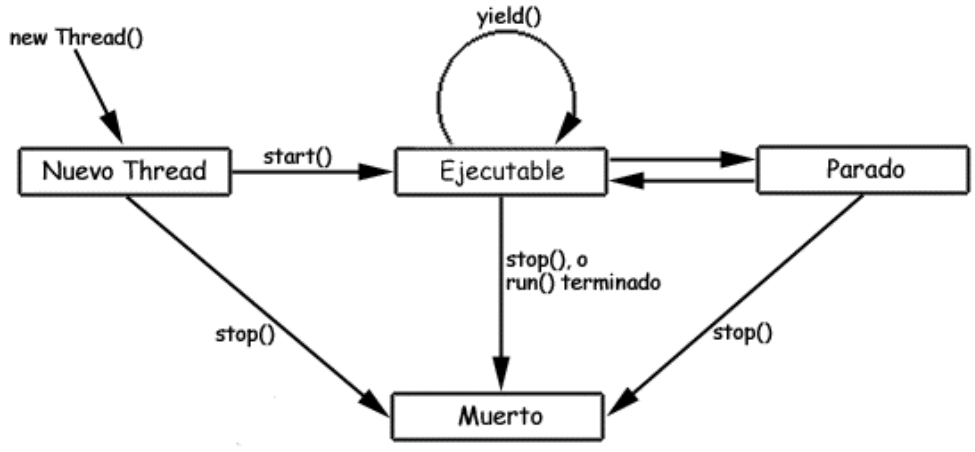

Figura 4. Estados de un hilo en Java.

La ventaja que proporciona el uso de hilos en este proyecto es la de poder ejecutar varias tareas de forma simultánea, se calcularan medidas sobre quince puntos en simultaneo con el trabajo de un servidor que recogerá esos datos y los enviara otro programa independiente. Estos conceptos se afianzan en el capítulo 4.1.

Los estados de un hielo son: *nuevo, ejecutable, parado* y *muerto.* Los métodos que controlan el estado de los hilos y su correspondencia se pueden ver en la figura 4.

2.6. Arquitectura cliente - servidor

La arquitectura cliente-servidor es un modelo de aplicación distribuida en el que existen dos sujetos, los servidores que proveen de recursos o servicios y los clientes que realizan peticiones a otro programa, concretamente a los servidores. Esta idea se puede aplicar a programas que se ejecuten en una maquina pero es más útil en sistemas multiusuario a través de una red.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores, con la ventaja de separar responsabilidades y facilita el diseño y gestión del sistema. La separación entre cliente y servidor es una separación de tipo lógica, donde el servidor no se ejecuta necesariamente sobre un solo programa. En este proyecto el servidor será ejecutado sobre un hilo Java implementado por una clase dentro del proceso principal.

La red cliente-servidor es aquella red de comunicaciones en la que todos los clientes están conectados a un servidor, en el que se centralizan los recursos y que los pone a disposición de los clientes. Esto significa que todas las gestiones que se realizan se concretan en el servidor, en el se dispone de los archivos o datos que requieren los clientes. Las redes pueden ser *LAN* (Wi-Fi, Ethernet) o *WAN* (VPNs, túneles securizados).

Existen varios modelos C/S. A continuación se comentara sobre el modelo más sencillo llamado de “dos capas”.

Las dos capas que dan nombre a este modelo se refieren a la parte que implementa la interfaz (cliente) y la parte donde se encuentra el gestor de bases de datos (servidor) y trata las peticiones. A la hora de implementar un modelo se ha elegido este por sus ventajas: Se mantiene una conexión persistente con la base de datos. Al tener peticiones de datos muy seguidas en el tiempo es más eficiente mantener la conexión durante toda la vida del programa. Este modelo también implica desventajas, al

desestimar usar una capa intermedia o *middleware* la parte cliente y la servidora tienen que conocer los controladores necesarios para el intercambio de datos. En este caso los controladores serán los referidos a la librería *Processing.net* (véase capítulo 5).

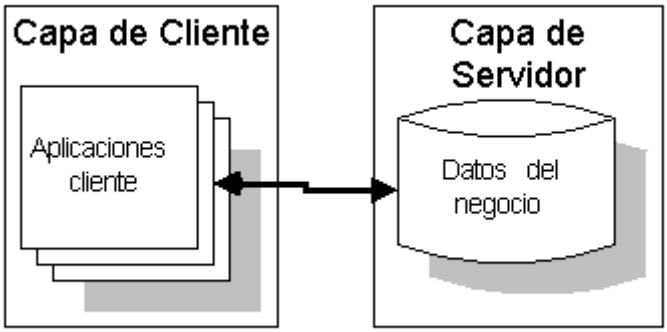

Figura 5. Modelo C/S de dos capas[15].

[15] Imagen obtenida de: http://docente.ucol.mx/sadanary/public_html/bd/cs_archivos/image004.gif

Descripción experimental

Capitulo 3. Obtención y uso del skeleton

OpenNI tiene la capacidad de procesar las imágenes con el fin de detectar y rastrear a las personas. En lugar de tener que analizar todos los puntos de la escena para comprobar su posición, se podrá simplemente acceder a la posición de cada parte del cuerpo de cada usuario que OpenNI ha reconocido. Una vez OpenNI ha detectado un usuario, devolverá la posición de cada una de las articulaciones[16] del usuario: cabeza, cuello, hombros, codos, manos, torso, las caderas, las rodillas, los pies. Para este proyecto esto es exactamente lo que se necesita. Podemos rastrear los movimientos del cuerpo a través del tiempo y compararlos, y podemos medir los valores de estas.

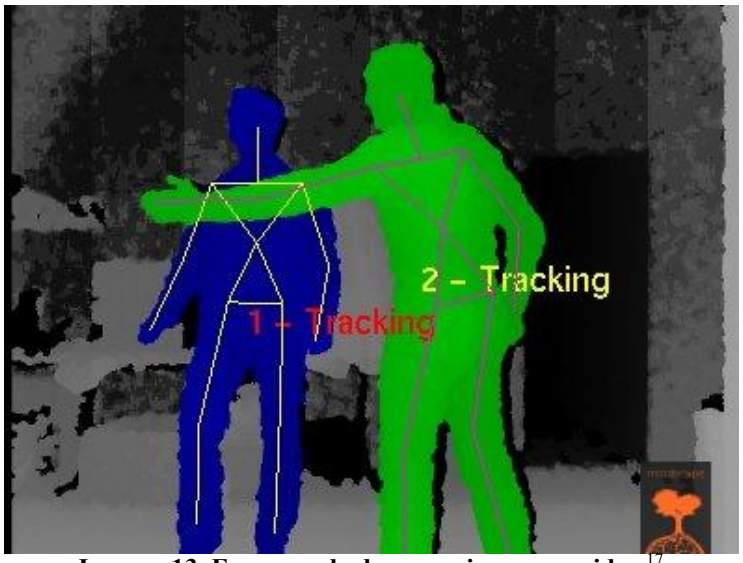

Imagen 13. Esquema de dos usuarios reconocidos.[17]

Como se ya se ha dicho los datos del esqueleto se generan mediante el procesamiento de la imagen capturada por Kinect. OpenNI también puede extraer más información de la imagen además de las ubicaciones de las articulaciones del usuario. En primer lugar, OpenNI puede averiguar qué píxeles de la imagen de fondo representan a las personas y cuáles representan el fondo de la escena. Esta información estará disponible en el momento que OpenNI detecta que una persona ha entrado en la

[16] Nota: No todas las articulaciones son en el sentido anatómico; OpenNI utiliza el término articulación o punto de unión como término para referirse a todos los puntos del cuerpo de un usuario que es capaz de rastrear si son o no uniones reales.
[17] Imagen obtenida de http://www.signalprocessingsociety.org/

escena. El algoritmo detecta si una persona está presente y realiza un seguimiento continuo a medida que se mueve dentro del campo de visión de la cámara. Así que es posible distinguir varios usuarios entre sí y diferenciarlos del fondo (imagen 13). OpenNI mapea esta información sobre los usuarios en una matriz de números que corresponden a cada píxel de la imagen de profundidad. El valor añadido para cada píxel será 0 si el píxel es parte del fondo y si el píxel es parte de un usuario el valor del mapa será el identificador del usuario: uno, dos, tres, etc.

El esqueleto se puede representar mediante la figura *stickman* que sigue la postura del usuario, esto mostrara todas las articulaciones a las que se tiene acceso y que partes del cuerpo del usuario representan.

A continuación se tratara el proceso de calibración que tenemos que usar, por qué existe este procedimiento y la forma de trabajar con él. SimpleOpenNI proporciona sus propios métodos que permiten ejecutar código en los momentos críticos del proceso de seguimiento de los usuarios: cuando un nuevo usuario se descubre, cuando se inicia el seguimiento por primera vez, cuando se pierde a un usuario, etc.

3.1. Conceptos sobre skeleton

En esta parte se tratara el concepto de *skeleton* o anatomía según OpenNI. Se verán qué articulaciones están disponibles mediante la biblioteca y cómo se accede a ellas. Se usara parte del código del archivo Skeleton.pde (Anexo A) para ilustrar la explicación.

En la imagen 14 se muestra el patrón de conexiones estándar. Se utiliza por PrimeSense en sus aplicaciones de ejemplo. A simple vista parece un esbozo del esqueleto humano pero hay que puntualizar ciertas partes. Se conectan las manos hasta los codos y los codos a los hombros de forma fidedigna a la anatomía del usuario. Lo mismo vale para las piernas: la conexión de los pies a las rodillas y las rodillas a la cadera. Sin embargo las cosas difieren un tanto de la idea habitual de la anatomía humana. El torso parece estar compuesto de dos triángulos que se unen en el centro del pecho. Además, los hombros están conectados directamente al cuello y el cuello no está en absoluto conectado a las caderas. En un esqueleto humano real la cabeza está

conectada a la columna vertebral que desciende a través del cuello y finalmente se conecta a la pelvis, que constituyen las caderas.

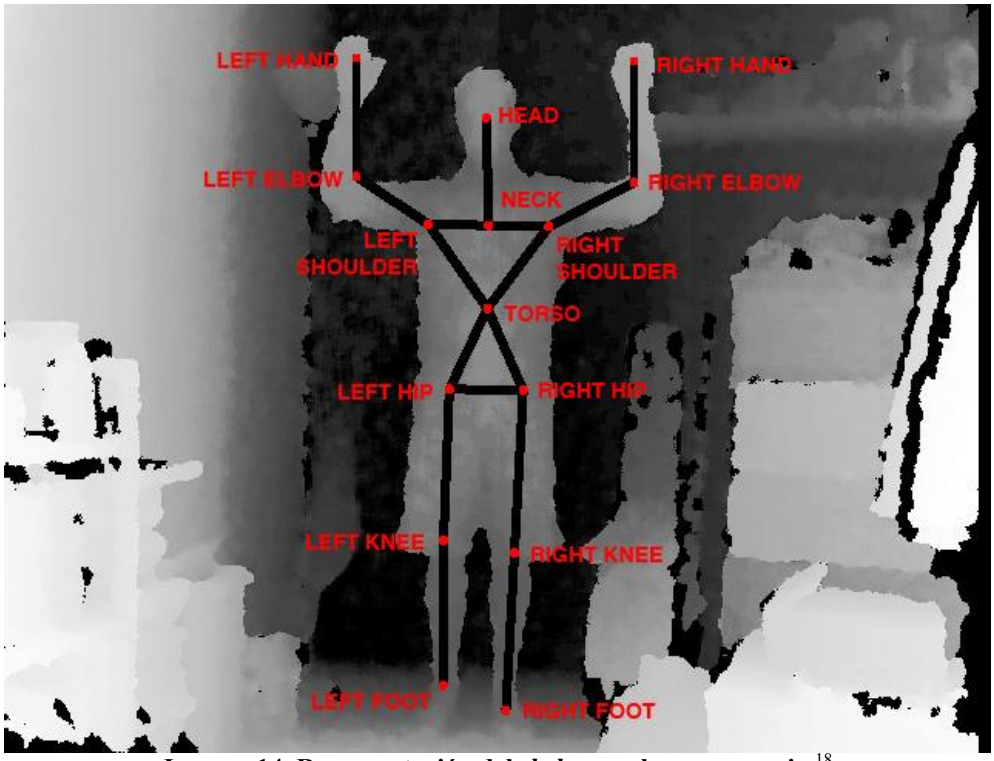

Imagen 14. Representación del *skeleton* sobre un usuario.[18]

Se quiere aclarar con estas diferencias entre modelos anatómicos que las articulaciones y extremidades reconocidas por OpenNI pretenden ser útiles para la creación de aplicaciones interactivas, no para ser anatómicamente correcto. Aun con estas mínimas diferencias este esquema es totalmente útil para nuestro objetivo por su simpleza y rapidez de procesamiento.

3.1.1. Limitaciones del modelo skeleton

Lo primero a destacar es que algunas articulaciones no se representan con total exactitud. Por ejemplo los codos están la vez un poco por encima y más cerca del cuerpo. Por otro lado, los hombros y el cuello están sensiblemente más bajos, el hombro derecho y el hombro izquierdo se encuentra mucho más cerca del pecho (ver imagen 14). Este es el límite en la precisión del sistema.

[18] Imagen tomada de Making Things See – Greg Boreestein

La variable *torso* resulta ser uno de los puntos más útiles para seguir. Hay muchas situaciones en las que es posible que se desee simplemente un seguimiento a un usuario sin preocuparse por sus extremidades o la cabeza, es decir, como un todo. Esta variable es probablemente la articulación más visible. A medida que el usuario gira o se mueve alrededor de la escena, las manos, los codos, las rodillas y los pies pueden entrar fácilmente en sombra, es decir, estar detrás de otras partes o detrás del mismo tronco. Dicho de otra manera, el nivel de "confianza" o valor de *confidence* (ver capitulo 4) asociado con estas partes es muy variable. Sin embargo OpenNI probablemente conocerá la posición del torso en todo momento con un alto valor de *confidence*.

3.2. Calibración

Para reconocer a una persona y comenzar el seguimiento el algoritmo que implementa OpenNI necesita que el usuario se coloque en una pose definida. En concreto hay que estar con los pies juntos y los brazos levantados por encima de los hombros a los lados de la cabeza. Esta postura es conocida por varios nombres. En la propia documentación de PrimeSense se le llama pose PSI. Otros diseñadores se refieren a ella como "el hombre inocente" debido a su gran parecido a la posición de una individuo si alguien le apunta con un arma. La estandarización de la pose hace que OpenNI pueda medir la longitud de los brazos, el tamaño del torso y el ancho. OpenNI usará esta información para hacer el seguimiento de todas las articulaciones del esqueleto de forma rápida y precisa.

Esta calibración plantea ciertos problemas. Si se busca el seguimiento de personas mientras realizan sus actividades normales, la necesidad de que se detengan y realizar una calibración explícita es un obstáculo difícil. Pero por otro lado, hay situaciones donde la posibilidad de seguimiento sin una calibración explícita, por ejemplo con cámaras *bodytracking,* que pueden observar todos los movimientos de un individuo sin su conocimiento en un espacio público planta problemas éticos y de intimidad en la sociedad.

Sin embargo OpenNI tiene la capacidad de seguimiento de usuarios sin la necesidad de calibración explícita bajo ciertas circunstancias. OpenNI puede grabar los datos de calibración de un solo usuario y luego utilizar esto para calibrar otros usuarios

siempre y cuando compartan una anatomía similar. Esta técnica no funcionará en todas las situaciones, por ejemplo, si se espera que una amplia gama de personas, incluyendo niños y adultos, hombres y mujeres, etc. En este proyecto no se aplica esta técnica.

3.2.1. Flujo de calibración

El diagrama 4.1 muestra el flujo de estados y las llamadas a los métodos de OpenNI que se usara para continuar el proceso de calibración.

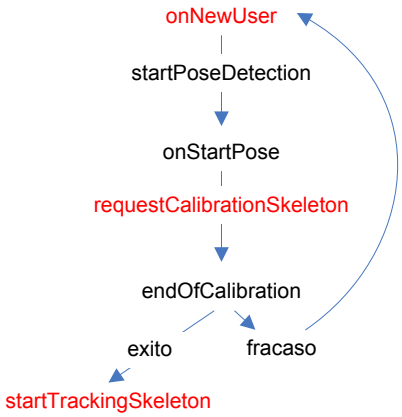

Figura 6. Proceso de calibración.

Como se puede ver es un proceso secuencial de llamadas a los métodos que ofrece OpenNI. La parte del código que se escriben en el programa principal se muestra más adelante. Las tres etapas para rastrear a una persona son la detección básica de usuario, la calibración y el reconocimiento completo. Cuando el programa se ejecuta por primera vez OpenNI no estará rastreando ningún usuario. El proceso comienza cuando un usuario entra en la visión de la cámara de profundidad. Cuando aparece el usuario, OpenNI reconoce que está presente. En este punto el usuario no se ha calibrado y sus conjuntos de datos no están disponibles. OpenNI básicamente tiene el "presentimiento" de que alguien se mueve dentro de la visión de la cámara.

```
void onNewUser(int userId) {                          (1)
  println("start pose detection");
  kinect.startPoseDetection("Psi", userId);           (2)
}
void onEndCalibration(int userId, boolean successful)(6){
  if (successful) {
    println(" User calibrated !!!");
    kinect.startTrackingSkeleton(userId);             (7)
    startTime = clock.startTime();
  }
  else {
    println(" Failed to calibrate user !!!");
    kinect.startPoseDetection("Psi", userId);         (2.1)
  }
}
void onStartPose(String pose, int userId) {           (3)
  println("Started pose for user");
  kinect.stopPoseDetection(userId);                   (4)
  kinect.requestCalibrationSkeleton(userId, true);(5)
}
```

Código 3.1. *Callbacks* o métodos de calibración.

Lo primero a destacar es que *onNewUser* (1) toma un argumento, un entero que representa el ID del usuario detectado. OpenNI puede realizar un seguimiento a varias personas al mismo tiempo. Todos los usuarios tienen identificadores únicos para mantenerlos diferenciados de los demás usuarios. El método *onNewUser* (1) es invocado por OpenNI y dentro de este se llama a su vez al método de seguimiento *startPoseDetection* (2) al que se le pasan dos variables, el nombre de la pose y el ID del usuario. Esta llamada hace que OpenNI compruebe si el usuario esta en pose de calibración. Una vez que se ha detectado un usuario la lista de los usuarios dentro del método *draw* aumentara. Sin embargo, los puntos de unión o articulaciones no estarán disponibles para estos usuarios hasta que el proceso de calibración se haya completado.

Tan pronto como el usuario asuma la pose OpenNI llamará función *onStartPose* (3) del programa pasando el nombre de la pose y el número de usuario. A diferencia de *onNewUser*, *onStartPose* toma dos argumentos. Sigue pasando el ID del usuario al que está realizando el seguimiento, pero ahora el primer argumento es el nombre de la postura. Mediante *onNewUser* OpenNI conoce la postura que se va a pedir que realice el usuario, postura Psi. Por esta razón el argumento: *startPoseDetection ("PSI", ID de usuario)*. Hipotéticamente, este sistema puede funcionar con otros patrones de

calibración además de la pose de Psi. A continuación mediante el método *stopPoseDetection* (4) se para la detección de la pose, ya que si no OpenNI continuaría con ese proceso haciendo que se llamara repetitivamente a *startPoseDetection*. Para continuar con el proceso de calibración se invoca al método *requestCalibrationSkeleton* (5), pasando el *IDuser*. Este proceso detecta el esqueleto como lo conoce OpenNI (véase nota 15 capitulo 3) comenzado la búsqueda de todas las posiciones o articulaciones de aquí en adelante. Al estar el usuario en una posición conocida durante la calibración, tal como la postura Psi, OpenNI es capaz de ajustar todo para adaptarse al tipo de cuerpo del usuario actual ya que no todos los usuarios tienen la misma complexión. El trabajo que realiza *onStartPose* no afecta a los resultados de la función *draw* en absoluto.

Como *onStartPose*, *onEndCalibration* pasa dos argumentos. A diferencia de *onStartPose* en este momento de la secuencia realmente se necesita la información contenida en estos dos argumentos. El primer argumento es el identificador de usuario. Al igual que se ha visto en las dos funciones anteriores esto permite distinguir que el usuario acaba de terminar la calibración en el caso que varios usuarios estén en la escena. Al igual que antes, se pasara este ID de usuario a través de otras funciones OpenNI para completar o continuar con el proceso de calibración.

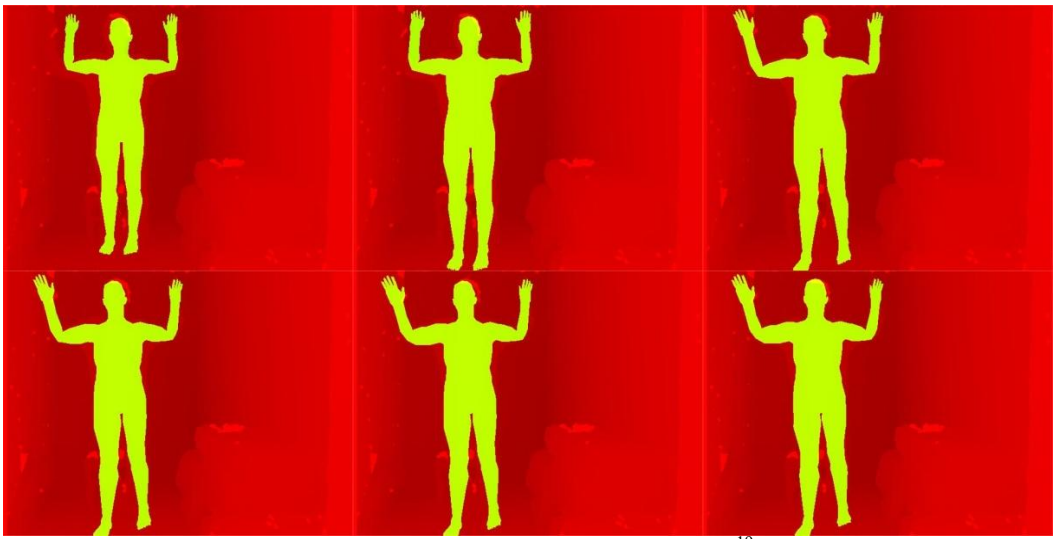

Imagen 15. Pose PSI para calibración.[19]

[19] Imagen obtenida de mutatedfantasy.wordpress.com

OpenNI utiliza la llamada a *onEndCalibration* (6) del programa para anunciar que ha terminado el método anterior (5) mediante los parámetros *IDuser* y la variable booleana *successful*. Esto es así ya que el proceso de calibración no siempre tiene éxito. La calibración también puede fallar, esto puede suceder por varias razones. Por ejemplo, un usuario podría estar situado fuera del rango de profundidad de imagen de tal manera que el Kinect no puede ver lo suficiente de su cuerpo o si una cantidad suficiente del cuerpo del usuario esta en sombra. Con menor frecuencia la calibración puede fallar si el usuario tiene una complexión inusual. Dentro de este método (6) si ha tenido éxito se comienza el seguimiento del esqueleto llamando a *kinect.startTrackingSkeleton(IDuser)* (7). Después de llamar a esta función, los datos de los puntos estarán disponibles para el usuario dentro de dibujar (método *draw*). Una vez completado el proceso de calibrado no se disparará de nuevo durante la ejecución del programa a menos que un nuevo usuario aparece en la escena (o el primer usuario salga de la escena y deba ser recalibrado). Si la calibración falla, se reiniciara el proceso llamando *startPoseDetection* (2.1) de nuevo de forma cíclica. Este bucle (figura 6) de *startPoseDetection* (4) hasta *onEndCalibration* (6) continuara indefinidamente hasta que calibramos correctamente el usuario o paremos el programa.

3.3. Obtención de usuarios

Dentro del método nativo de Processing *draw()* codificado en el programa principal (capitulo 4) se encuentran la manera de pedir a OpenNI una lista de todos los usuarios a los que está realizando un seguimiento. OpenNI representa a los usuarios como enteros, a cada usuario se le asigna un único número como su ID de usuario. Una vez que se ha detectado un usuario, el ID que se recibió en *onNewUser* (código 3.1) se mostrará en la lista de usuarios que OpenNI proporciona (código 3.2).

La primera línea (1) se crea una variable *lista_usuarios* para almacenar los identificadores de usuario. Esta variable se declara como *IntVector*. Este es un tipo especial de variable proporcionada por OpenNI específicamente para almacenar listas de números enteros que representan los ID de usuario. En la segunda línea (2) la función *GetUsers* añadirá todos los ID de usuario que OpenNI ha detectado en el *IntVector* que se pasa como argumento. Hay que tener en cuenta que no hay ningún valor de vuelta de esta función. No se asigna la variable a la salida de la función. Esta es una diferencia

fundamental en la forma convencional de paso de parámetros. Después de esta línea, *lista_usuarios* contendrá cualquier ID de usuario que obtenga OpenNI.

```
IntVector userList = new IntVector();          (1)
kinect.getUsers(userList);                     (2)

if (userList.size() > 0) {                     (3)
   int userId = userList.get(0);               (4)
   insertarUsuario(userId);

   if (kinect.isTrackingSkeleton(userId)){     (5)
   ...
   }
...
}
```

Código 3.2. Lista de usuarios.

Se ha detectado la presencia de un usuario pero aun no se tiene acceso a la posición de las articulaciones hasta que se complete el proceso de calibración. Dentro del método *draw()* el código que tiene acceso a las posiciones de los usuarios está rodeada por dos sentencias *if*. Sólo se podrá acceder a esas posiciones si se sigue un usuario y se ha completado el proceso calibración. Ahora se obtiene una *lista_usuarios* que contiene un identificador de usuario.

La primera sentencia *if* (3) devolverá *true.* Por tanto *lista_usuarios* contendrá un entero que representa el ID del usuario que se ha detectado. Se extrae el ID (4) del primer elemento de *IntVector* llamando a *userList.get(0)* donde 0 es la posición inicial. Este ID de usuario se definirá con el mismo número entero que se pasó a la función *onNewUser*. Para terminar con este proceso se comprueba si el usuario se ha calibrado correctamente. Para ello, hay que hacer una llamada a *kinect.isTrackingSkeleton* pasando el ID del usuario como parámetro. Esta función devuelve un resultado positivo sólo si el proceso de calibración se ha completado y el conjunto de puntos está disponible para el usuario. Sin embargo, si no ha terminado el proceso de calibración se devolverá *false* y el flujo finalizará sin acceder a la información conjunta repitiéndose la secuencia de *draw* (véase el funcionamiento del método en el capítulo 4).

Una vez codificadas estas líneas ya se trabajara con el ID del usuario. Esto permite diferenciar usuarios y actuar en consecuencia. Como nota mencionar que este

proyecto no se ha implementado para más de un usuario (véase capitulo 6.2 Futuras mejoras).

3.4. Comprobaciones de la calibración

Para garantizar la fiabilidad de la calibración se han realizado pruebas mediante programas auxiliares que no implementan la lógica del proyecto si no que permiten obtener medidas y compáralas con referencias reales conocidas. Conociendo una medida real en este caso entre las dos manos (imagen derecha) de un sujeto de pruebas, se obtendrá la misma medida para el mismo segmento mediante el programa de prueba.

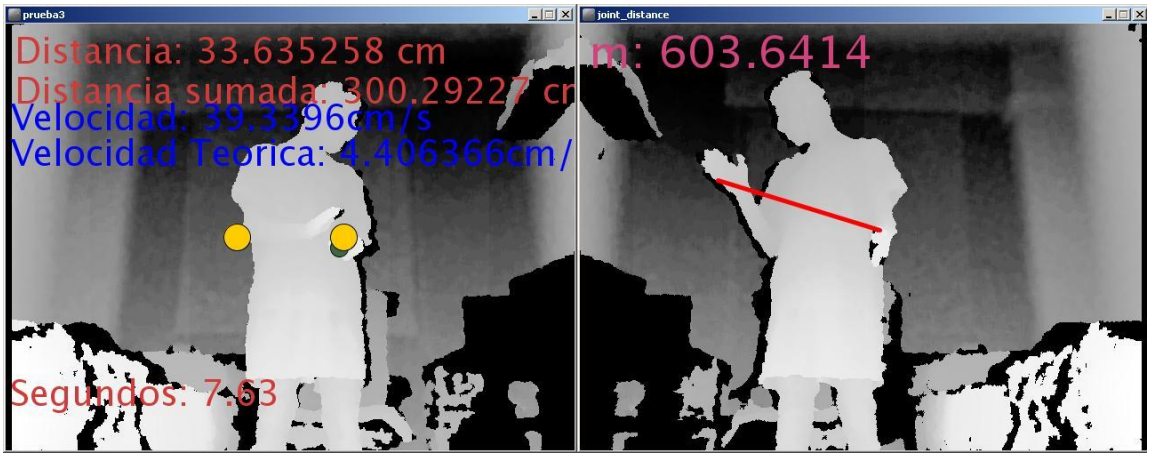

Imagen 16. Pruebas de calibración.

Kinect mediante su cámara de IR sabe en qué posición respecto a ella (origen de coordenadas) nos encontramos. Véase Figura 7. Conociendo la posición, dentro de la habitación, de los puntos seleccionados (las dos manos) y gracias al método nativo *differenceVector.mag()* se calcula la distancia entre los puntos en el plano x, y, z que crea la habitación. Antes de iniciar el programa se mide con un metro la distancia entre puntos. Al volver a medir la distancia anterior mediante el programa se compara el resultado con las medidas reales y se aprecia la precisión.

Imagen 17. Comprobación de la medida mediante el metro.

```
if (userList.size() > 0) {
  int userId = userList.get(0);

  if ( kinect.isTrackingSkeleton(userId)) {

    PVector leftHand = new PVector();
    PVector rightHand = new PVector();

    kinect.getJointPositionSkeleton(userId,
                      SimpleOpenNI.SKEL_LEFT_HAND, leftHand);
    kinect.getJointPositionSkeleton(userId,
                      SimpleOpenNI.SKEL_RIGHT_HAND, rightHand);

    // calculate difference
   PVector differenceVector = PVector.sub(leftHand, rightHand);

    // calculate the distance and direction
    magnitude = differenceVector.mag();
    differenceVector.normalize();

     //INTERPOLAR
    lastMagnitude = lerp(magnitude,lastMagnitude,0.3f);

    // draw a line between the two hands
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_HAND,
                                  SimpleOpenNI.SKEL_RIGHT_HAND);
```

Código 3.3. Parte de la codificación de los programas de prueba.

Una versión mejorada del programa de pruebas suma la distancia como lo haría el programa final. Para comprobar que la suma es correcta se hacen movimientos fijados y medidos anterior mente. Se desplaza el punto de control, esta vez una mano, a lo largo

de dos puntos fijos cuya distancia es conocida. La suma de esos caminos obtenida por el programa se compara con la real.

Sin duda alguna estas comprobaciones son el proceso más complejo por el que pasa el programa final. La lógica fundamental reside en tener una buena calibración y obtener las medidas de distancia que serán básicas para calcularas las demás con la mayor precisión posible.

3.5. Implementación del skeleton

Para dibujar el *skeleton* en pantalla se ha creado una clase independiente a la lógica principal del programa. Este archivo llamado *Skeleton.pde* (Anexo A) se implementa dentro del programa principal como una clase *Skeleton* (gracias a la herencia de Java en Processing). El objeto que implementa el *skeleton* se llama una vez (código 3.4) en el flujo principal del programa o método *draw()*. La llamada se encuentra dentro de la condición *if kinect.isTrackingSkeleton(userId)* ya que solo se podrá dibujar un usuario cuando este esté disponible para el seguimiento. El método necesita el ID de usuario para saber a qué persona dibujar (en el caso de haber más de una a la vez).

```
if (kinect.isTrackingSkeleton(userId)){
  iniciarHilos();
  sk.drawSkeleton(userId);          //Llamada
  ...
}
```

Código 3.4 Llamada a *drawSkeleton* dentro de *draw*.

El Anexo A muestra la clase *Skeleton* completa. Es un código sencillo donde se repiten dos grandes bloques: dibujar las uniones, y dibujar los puntos. Su objetivo es simplemente dibujar en pantalla las uniones de las articulaciones.

```
void drawSkeleton(int userId) {
  stroke(0);
  strokeWeight(5);
  kinect.drawLimb(userId,                                    (1)

     SimpleOpenNI.SKEL_HEAD,SimpleOpenNI.SKEL_NECK);
...
  noStroke();
  fill(255, 0, 0);
  drawJoint(userId, SimpleOpenNI.SKEL_HEAD);                 (2)
...
}
```

Código 3.5. Extracto del código Skeleton.pde

La necesidad de trazar articulaciones es tan común que SimpleOpenNI ya ofrece un método para ello. La función *drawSkeleton* es la de dibujar el esqueleto en pantalla, para ellos se basa en los métodos nativos de OpenNi. Las primeras líneas de *drawSkeleton* configuran mediante métodos de Processing el color y grosor del trazo. A continuación mediante el método nativo (1) *kinect.drawLimb(userId, SimpleOpenNI.SKEL_HEAD, SimpleOpenNI.SKEL_NECK)* se dibuja el trazo entre la articulación de la cabeza y el cuello. Esto se repetirá secuencialmente por cada trazo de unión que se necesite. *kinect.drawLimb* toma tres argumentos: el ID del usuario seguido y dos constantes que identifican articulaciones en el esqueleto. Indicamos las articulaciones mediante un sistema similar al de estos ID de usuario numérico. Sin embargo, en lugar de variar con cada usuario que detectamos las identificaciones de las articulaciones son las mismas en todos. SimpleOpenNI incluye constantes para todas las articulaciones que es capaz de reconocer. Dentro *drawLimb*, SimpleOpenNI utiliza *getJointPositionSkeleton* otro método nativo para acceder a la ubicación de las articulaciones. El resultado final con todos los segmentos dibujados corresponde a las imágenes 14 y 18.

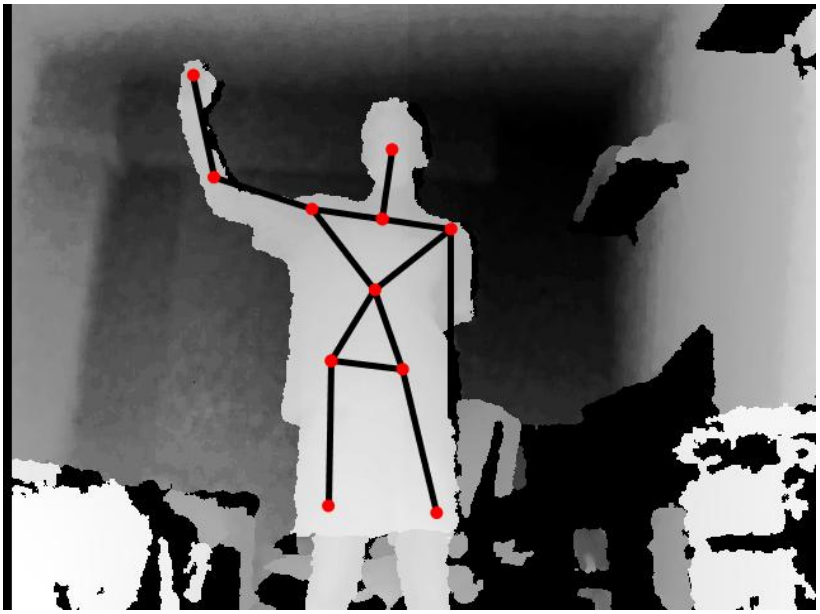

Imagen 18. Las partes ocultas no se dibujan.

Una buena característica adicional de *drawLimb* es que toma en cuenta el valor de *confidence* para las articulaciones que se dibujan. Si cualquiera de los miembros tiene un valor bajo de *confidence*, *drawLimb* no dibujara el trazo. Esto evita que se dibujen miembros errantes. En la imagen 17 los miembros correspondientes a las partes del cuerpo del usuario que se oculta simplemente desaparecen. Por tanto estos miembros como se verá más adelante tendrán información cero.

Respecto a las articulaciones, SimpleOpenNI no proporciona una función para la visualización. Así pues, se ha escrito una propia *drawJoint*. La llamada a esta función se utiliza dentro del método *drawSkeleton* (código 3.5 (2)). Solo se necesitan dos parámetros: ID usuario e ID de la unión. Por tanto este método se usara de formar secuencial una vez por cada articulación que se quiera dibujar, tal y como se hace con *drawLimb*.

```
void drawJoint(int userId, int jointID) {              (1)
    PVector joint = new PVector();
    float confidence =                                 (2)
      kinect.getJointPositionSkeleton(userId, jointID, joint);

   if (confidence < 0.7) {
      return;                                          (3)
   }

   PVector convertedJoint = new PVector();
   kinect.convertRealWorldToProjective(joint,          (4)
                                       convertedJoint);
   ellipse(convertedJoint.x, convertedJoint.y, 10, 10);
}
```

Código 3.6. Código extraído de Skeleton.pde correspondiente al método creado *drawJoint.*

Los pasos a seguir son los siguientes: se crea una variable *PVector* (tipo de datos de Processing) a la que se pasara la información de la articulación solicitada. Mediante el método proporcionado por OpenNI *getJointPositionSkeleton(userID, jointID)* (2) obtenemos la posición de la articulación. El resultado que devuelve el método lo guardamos en una variable de tipo *float* llamada *confidence*. Como se ha dicho antes solo dibujaremos las articulaciones que tengan un parámetro de *confidence* alto, para eso si (3) el valor es menor que 0,7 saldrá del método cancelando el flujo del programa y volviendo a *drawSkeleton.* Una vez asegurando que la articulación tiene un nivel mínimo de visibilidad se debe convertir la posición (2) a coordenadas proyectivas para que coincida con imagen de fondo, y luego, por último, usando el método *ellipse* de Processing se dibuja una elipse sobre la base de esas coordenadas convertidas. En el siguiente punto se detalla el porqué de esta conversión.

3.5.1. Coordenadas de los puntos de unión

Mediante los métodos correspondientes (código 3.6 (4)) OpenNI nos dará la posición del punto que se necesite en coordenadas reales. Estas son las coordenadas que corresponden a la posición física del cuerpo respecto a la habitación. Sin embargo, en el caso que queramos plasmar la articulación en una imagen de dos dimensiones y no trabajar con los valores se debe traducir en coordenadas proyectivas que coincidan con la profundidad de la imagen. Se utiliza una de las funciones de SimpleOpenNI hacer esto *convertRealWorldToProjective*. Una vez que tenemos las coordenadas se pueden utilizar para mostrar a posición de la articulación.

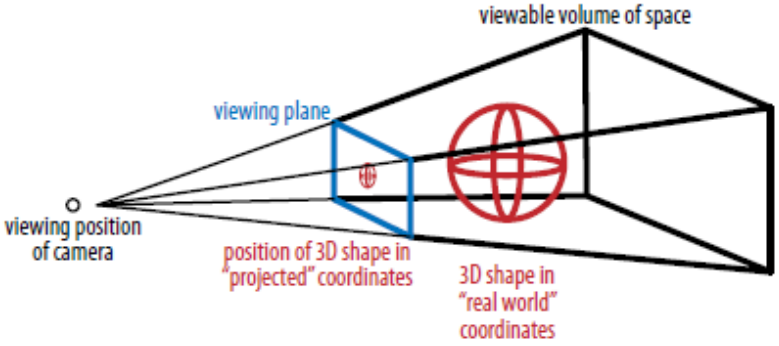

Figura 7. Representación de coordenadas reales y proyectivas.

Capitulo 4. Medidas sobre el usuario

En este capítulo se tratara la estructura del programa mediante el código implementado y comentando los aspectos más interesantes de este.

El lógica principal reside en 15 hilos, uno por cada parte del cuerpo con las que queremos trabajar. Estos hilos se crearan en la clase principal que hereda de *PApplet* y se iniciaran una vez se tenga un usuario calibrado (Capitulo 3). Antes de esto lanzar un hilo donde se encontrara el servidor (Capitulo 5) programado en base a la librería Network de Processing.

4.1. Creación y vida de los hilos

En el diagrama 4.1 se muestra la vida de los hilos principales. El hilo principal en el que comienza el programa lanzara un servidor donde se conectarán los clientes. El servidor esperara a recibir los datos de los hilos "productores" para enviar un paquete que formarán los datos de los 15 puntos.

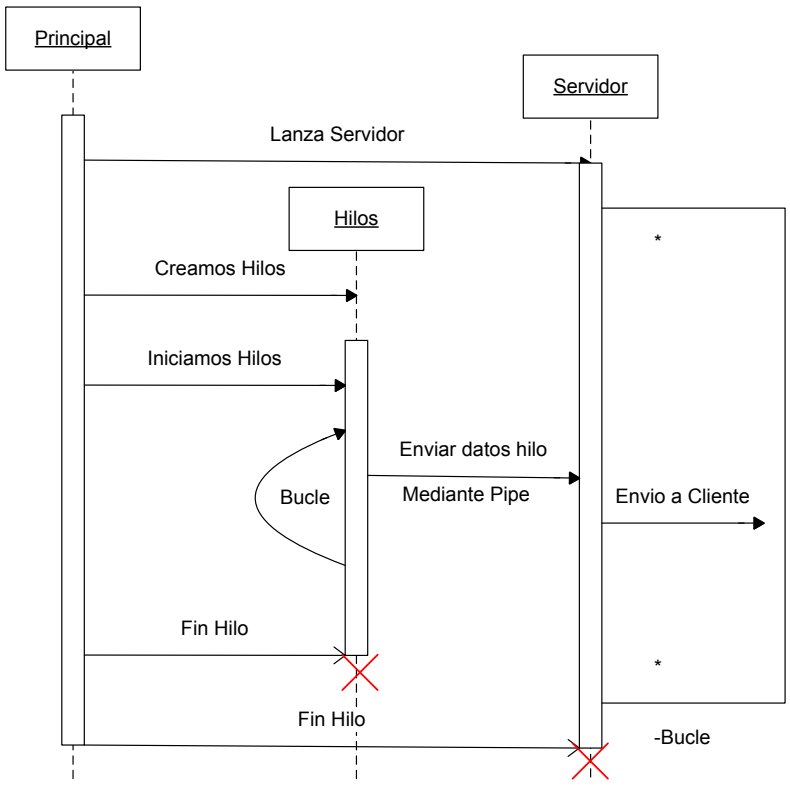

Diagrama 4.1. Vida de los hilos principales.

Antes de todo esto se tienen que crear e iniciar los hilos. El programa principal crea los hilos (uno por cada punto que queramos). Estos hilos mediante una tubería o *pipe* cada cierto tiempo (medio en segundos). Calcularán las medidas cinemáticas de la articulación o parte del cuerpo y las enviaran al servidor. Una vez el programa principal detecta la orden de fin envía señales a los hilos para que terminen.

Con esta estructura se gana en rapidez a la hora de hacer cálculos, los 15 hilos trabajan en paralelo mientras que el servidor también en paralelo espera a recibir los datos. Ininterrumpidamente (cada 30fps) el programa principal gracias a la clase *Skeleton* irá pintando en pantalla los puntos en su lugar correspondiente (capitulo 3).

```
void crearHilos() {
  String lines[] = loadStrings("BaseDatos.txt"); (1)
  for (int i=0; i<NHILOS; i++) {
    String words[] = lines[i].split(",");
    aHilos[i] = new Hilo(kinect,                  (2)
                Integer.parseInt(words[0]), tiempo, pipe);
    aHilos[i].setName(words[2]);
  }
}
```

Código 4.1. Creación de los hilos.

Con la ayuda de una base de datos en formato texto (1) donde se encuentra el id del punto del cuerpo que usa OpenNi, y el nombre de ese punto se crearan los hilos. El hilo para poder trabajar necesita la referencia de la cámara (variable *kinect*), el id del punto que trabajara, el tiempo en el que comienza (para sincronizar los hilos con el principal) y la tubería mediante envía los resultados al *Server*.

4.2. Clase principal

4.2.1. Diagrama de Flujo (método draw)

Nos centramos en el método *draw()* obviando la calibración (capitulo 3) y asumiendo que el servidor ha sido lanzado y está en funcionamiento. El programa principal mediante *draw()* una vez tenga al usuario localizados y calibrado iniciaría los hilos correspondientes a este uno por cada parte. Como se ha visto (capitulo 2.4.2.2) el método *draw()* se repetirá con una frecuencia de 30 veces por segundo.

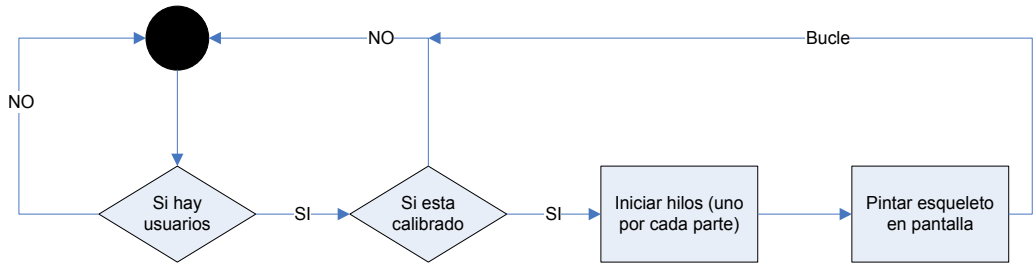

Diagrama 4.2. Flujo de la clase principal en el primer momento.

Hay que destacar que el inicio de los hilos no se repite. Solo se inician una vez. Esto se consigue mediante el código siguiente (código 4.2).

```
void iniciarHilos() {
  for (int i=0; i<NHILOS; i++) {
    if (aHilos[i].getState() == Thread.State.NEW) {
      aHilos[i].setStartTime(startTime);
      aHilos[i].start();
    }
  }
}
```

Código 4.2. Inicio de los hilos

Dentro del bucle mediante la sentencia *if* se interroga el estado del hilo. Si el hilo está en estado *NEW* es decir no se ha iniciado aún se procederá a lanzarlo en caso contrario la clase principal seguirá su curso (véase Figura 4).

4.2.2. Codificación

Esta es una clase heredada de *PApplet* donde se apoya Processing para iniciar el desarrollo de la aplicación.

```
import SimpleOpenNI.*;
import java.lang.Object.*;                    (1)
import processing.net.*;

final int NHILOS = 15;                        (2)
final int PUERTO = 5204;
```

Código 4.2.1. Clase principal.

En el código anterior (4.2.1) se encuentra las clases que se deben importar (1) y las constantes (2) con las que se trabajara. Los *import* son necesarios para trabajar con los objetos y funciones de las librerías SimpleOpenNi y Network (Processing).

```
SimpleOpenNI kinect;
Server server;
Servidor servidor;
Skeleton sk;
Reloj clock;
Pipe pipe;
Hilo[] aHilos = new Hilo[NHILOS]
```

Código 4.2.2. Clases principales.

A continuación se instancian los objetos de las clases principales. Kinect tendrá las funciones necesarias para obtener las posiciones de los puntos del cuerpo. *Server* es una clase de la librería de Network, proporciona la salida de datos mediante un servidor. *Servidor* es una clase que extiende de *thread*, es decir es un hilo (véase cap. 2.5) donde se implementaran los métodos de *server*. *Skeleton* es otra clase particular para este programa (Anexo A). *Reloj* tendrá dos métodos que manejan el tiempo. *Pipe* proporciona una tubería necesaria para la comunicación entre hilos. *aHilos* es una array de objetos *Hilo*, donde se encuentran los métodos que calcularán todas las variables necesarias.

```
float startTime;
float tiempo=500;
```

Código 4.2.3. Declaración de variables.

Las variables que trabajaran con el tiempo y harán que los hilos y el principal vayan sincronizados son *startTime* (comienzo del tiempo) y *tiempo* que será una variable que marcara la frecuencia a la que los hilos trabajaran.

```
void setup() {
  size(640, 480);                          (1)
  frameRate(30);
  kinect = new SimpleOpenNI(this);         (2)
  kinect.setMirror(true);
  kinect.enableDepth();
  kinect.enableRGB();
  kinect.alternativeViewPointDepthToImage();
  kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);

  clock = new Reloj();                     (3)
  pipe = new Pipe();
  sk = new Skeleton(kinect);

  server = new Server(this, PUERTO);
  servidor = new Servidor(pipe, server);
  servidor.start();

  crearHilos();
}
```

Código 4.2.4. Método *setup*.

En el método *setup(),* nativo de Processing (capitulo 2), se crean las variables que instanciadas anteriormente. En las primeras líneas se definen el tamaño de la ventana y la velocidad de refresco, en este caso 30fps (1).

El segundo bloque de llamadas corresponde a métodos de SimpleOpenNI. Es necesario activar la cámara de profundidad (2) *enableDepth()* y la cámara RGB *enableRGB(). setMirror(true)* simplemente invierte la imagen que capta la cámara de forma que el usuario se ve como reflejado en un espejo, de esta manera es más fácil coordinar los movimientos. Mediante *alternativeViewPointDepthToImage()* se alinean las dos imágenes, profundidad y color. En la pantalla se mostrarán las dos en paralelo, en una de ellas se proyectara el esqueleto. Para terminar se debe indicar al objeto SimpleOpenNi (variable *kinect*) que se busca calibrar el cuerpo entero del usuario, esto es así ya que como se menciona antes (capitulo 3) se pueden tener varios modelos de calibrado, en este caso es el completo *skel_profile_all.*

En el punto (3) se crean las clases auxiliares que se han creado, subrayar que a *Skeleton* se le pasa la variable *kinect* ya que tendrá que obtener datos mediante la cámara. En la parte final se inicializa el servidor y se le pasa al hilo *Servidor* junto a la

tubería (*pipe*). El último paso es crear los hilos que esperaran a ser lanzados o inicializados en la parte *draw()*.

```
void draw() {
  background(0);
  kinect.update();
  image(kinect.depthImage(), 0, 0);
  image(kinect.rgbImage(), 640, 0);

  IntVector userList = new IntVector();
  kinect.getUsers(userList);
  if (userList.size() > 0) {                    (1)
    int userId = userList.get(0);
    insertarUsuario(userId);

    if (kinect.isTrackingSkeleton(userId)) {    (2)
      iniciarHilos();
      sk.drawSkeleton(userId);                  (3)
      textSize(15);
      text("Segundos: " + clock.endTime(startTime)/1000
                                          +" s", 100, 30);
    }
  }
}
```

Código 4.2.5. Método *draw*.

El método *draw()* se repetirá con una frecuencia tal y como se indica en *setup*. En este caso 30 veces por segundo.

Primero se harán las llamadas a métodos que dibujan en la pantalla las imágenes. Mediante *background(0)* se indica el color de fondo que tendrá la pantalla, el 0 es el valor para el negro. *Update()* es una llamada importante ya que hará que la pantalla se refresque en cada paso por el bucle *draw()*. Usando *image* función de Processing se sitúan las dos imágenes en sus coordenadas, imagen real y de profundidad. Una vez con las imágenes situadas y configuradas se entra en el flujo de *draw()*. En el diagrama 4.2 se ve esta parte del programa principal, aquí se explicara línea a línea.

Gracias a un vector de enteros la cámara mediante *getUsers(userList)* rellena la lista de usuarios que ha detectado. Siempre y cuando haya un usuario detectado (1) se seguirá con el siguiente paso que es introducir ese usuario en los hilos. Sin esta llamada los hilos no sabrían a que usuario seguir y cuál es el activo a la hora de calcular los

valores. Una vez el usuario ha finalizado la calibración (2) es hora de iniciar los hilos. Como se ha indicado antes esto solo ocurre una vez, esto es así ya que los hilos seguirán calculando los valores hasta que se diga lo contrario, por tanto no hace falta iniciarlos más veces (Código 4.2). Con los hilos ya iniciados y trabajando se pinta el esqueleto sobre la imagen y el tiempo que lleva el programa iniciado (3). Puntualizar que la llamada a *setText(15)* simplemente sirve para dar un tamaño a la letra que se escribe en pantalla.

4.2.2.1. La clase Reloj

La clase auxiliar *Reloj* cuyo código se encuentra en el Anexo B trabaja con las funciones de tiempo que nos ofrece Processing, en concreto *millis()*. Esta función da los milisegundos del reloj del ordenador. Una vez se tiene un usuario calibrado (Capitulo 3) se inicia un contador *startTime()* que corresponde con los milisegundos en ese momento. Siempre que se quiera saber cuánto ha trascurrido llamaremos a *endTime()* pasando los milisegundos de inicio. Este método nos devolverá en milisegundos el GAP de tiempo entre el inicio y el momento de la llamada. Si dividimos entre 1000 obtenemos los segundos que han pasado. Estos métodos se utilizan en más clases.

4.3. Clase Hilo

4.3.1. Diagrama de flujo

En la clase hilo reside la lógica del programa. Es donde se encontraran los métodos que permiten sacar los valores cinemáticos de las partes del cuerpo. Según el diagrama de flujo anterior (4.3) el hilo se repetirá infinitamente hasta que el programa principal no mande la señal de parada. Esto podría provocar un problema de sobrecarga en el sistema ya que hay que recodar que son quince los hilos que trabajan en paralelo a la velocidad de reloj del procesador. Para solucionarlo se dormirá el hilo (véase codificación 4.3). Una vez tenemos los datos del punto de control se envían al servidor mediante la tubería o *pipe*. Con ayuda del código se mostrara con más detalle la secuencia que sigue el hilo.

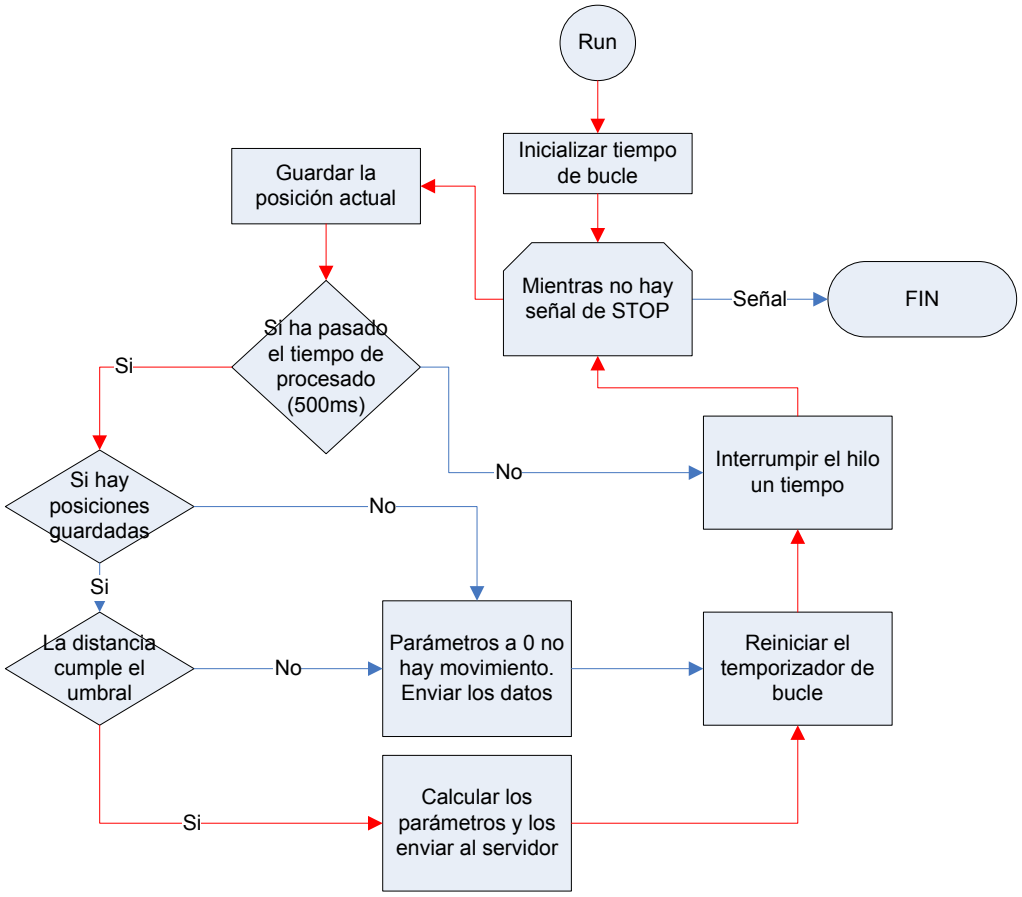

Diagrama 4.3. Diagrama de flujo para la clase hilo.

4.3.2. Codificación

El hilo como su nombre indica extiende de la clase *Thread* de Java. En el capítulo 2.5 se habla sobre el funcionamiento de los hilos en Java.

```
class Hilo extends Thread {

  SimpleOpenNI kinect;
  Reloj clock;
  Pipe pipe;
  JSONObject json;

  private boolean stop = false;

  private int jointID;
  private int userID;
  private ArrayList<PVector> frames;
  private int currentFrame = 0;

  private float tiempo;
  private float treshold = 15;
  private float startTime;
  private float loopTime;

  private float distancia;
  private float velocidad;
  private float aceleracion;

  PrintWriter outputVal;
  PrintWriter outputArray;
```

Código 4.3. Variables globales.

Como se ha dicho antes la clase *Hilo* extiende de *Thread* por tanto tendrá en el método *run()* el conjunto de sentencias que ejecutan la lógica del hilo. En un principio se encuentran las variables globales. Es necesario un objeto SimpleOpenNi que será el que nos de las posiciones del punto correspondiente, un objeto reloj para manejar el tiempo, una tubería para el envío de datos al hilo *Server* y una novedad respecto otras clases, un objeto de tipo *JSONObject* proporcionado por Processing. Este objeto se verá con profundidad en el tema 5.

El resto de variables serán utilizadas a lo largo de la ejecución del hilo. Destacar algunas. *frames* de tipo *ArrayList* es una lista indexada de *PVector*, guardara las posiciones por las que pasa el punto de control durante el tiempo que se esté analizando. *treshold* o umbral es una variable de control que hace descartar medidas menores a un valor fijo medido en milímetros. Esto se hace para que los saltos producidos por la precisión de la cámara sean obviados y no distorsionen las medidas (véase limitaciones cap.6). También distinguir entre *starTime* y *loopTime*. La primera variable tomara el

valor que se pase por medio del constructor al hilo y ese valor coincide con el tiempo de inicio del programa principal, *loopTime* marcara el tiempo que ha trascurrido desde el ultimo procesado de datos, en este caso cada 500 ms.

```
public Hilo(SimpleOpenNI tempContext, int tempJointID,
float tempTiempo, Pipe p) {
    kinect = tempContext;
    tiempo = tempTiempo;
    jointID = tempJointID;
    frames = new ArrayList();
    pipe = p;
    clock = new Reloj();
  }
```

Código 4.3.1. Constructor

El constructor del hilo se invoca al crear un nuevo hilo mediante el *new()*. En él se pasan los parámetros *tempContext* que hace referencia al objeto que implementa la Kinect, *tempJointID* el número de *joint* o punto de control que corresponde a este hilo, *tempTiempo* la frecuencia de procesado y *Pipe* la tubería de paso de datos entre hilos. Además de inicializar las variables que se han mencionado se crea la *ArrayList* de posiciones y la variable *Reloj*.

```
public void run() {
    JSONObject datosEnvio;                          (1)
    loopTime = startTime;

    while (!stop) {                                 (2)
      recordFrame();

      if (tiempo < clock.endTime(loopTime)) {       (3)
        if (frames.size()>1) {
          if (distancia() < treshold) {
            distancia = 0;
          }
          datosEnvio =                              (4)
                crearJSON(distancia,velocidad(),aceleracion());
          pipe.lanzar(datosEnvio);
          clearFrames();
        }else{
          datosEnvio = crearJSON(0,0,0);            (5)
          pipe.lanzar(datosEnvio);
        }
        loopTime = clock.startTime();               (6)
      }
      try {
        this.sleep(33); //33 milis = 1 frame
      }
      catch(InterruptedException ie) {
        println("SLEEP STOP");
      }
    }
  }
```

Código 4.3.2. Método *run()*.

El método comienza creando una variable auxiliar de tipo *JSONObject* que servirá para enviar los datos al servidor. (1) Seguidamente se inicializa el contador *startTime* que gestiona el tiempo de procesado. Para que el hilo no acabe en una primera pasada necesitamos introducir el código dentro de un bucle "infinito". Esto se consigue mediante la sentencia *while(stop).* (2) El hilo no acabara hasta que desde el programa principal se envíe una señal que haga que la variable *stop* cambie y el la condición no se cumpla. La primera sentencia dentro del bucle es *recordFrame()* (código 4.3.4), esta función almacena la posición en la que se encuentra el punto analizado. Una de los métodos fundamentales del hilo. El bucle se repetirá a una velocidad marcada por el procesador. Esto no interesa y para regular esta velocidad la se usa una variable *tiempo* (500ms) con el valor en ms del tiempo y sentencias de condición.

La primera condición *if* (3) se cumple cuando el *loopTime* es mayor a *tiempo*. La segunda sentencia *if* interroga si existen datos de posición. Esto sirve en el caso de que el punto se halle en sombra y no tengamos datos sobre él. La tercera condición se cumple en el caso de que la distancia calculada (se verá más adelante) supere el umbral. Si no lo superara la distancia sería menor y prácticamente 0. Una vez se cumple estas condiciones (4) y se tiene la distancia se calcula en base a esta la velocidad y la aceleración. El siguiente paso consiste en enviar los datos. Se verá más adelante (capitulo 5) como se crea la salida de datos en formato JSON. En el caso que no se tenga valores, por sombras por ejemplo, el servidor espera obtener medidas. En este caso se enviaran los valores a 0 (5). Una vez se ha terminado de procesar se reinicia el contador *loopTime* y se duerme al hilo *sleep()* (6). Con anterioridad se ha comentado que esta espera es para aliviar al sistema de una carga incensaría.

4.3.2.1. Métodos auxiliares

Se han creado tres métodos auxiliares básicos para calcular las variables necesarias.

Distancia() (1) devuelve el valor del recorrido que ha realizado el punto de control o articulación que estamos analizando. Esto se consigue mediante *PVector.dist(),* método de Processing que calcula la diferencia entre dos puntos del plano. Como se puede ver en el código se divide el trayecto final en dos segmentos. Desde el punto inicial al medio y del medio al final. La intuición hace que se busquen trayectos intermedios muy pequeños casi infinitesimales que sumados den la distancia exacta recorrida. Tras pruebas con diferentes puntos intermedios se ha optado por tres puntos. Como se dijo en los primeros capítulos Kinect tiene un rango de error que se asemeja al ruido de una señal, en el caso que usáramos n-puntos intermedios estaríamos sumando ese ruido dándonos medidas inexactas. Para un tiempo de medida de 0.5 segundos se ha comprobado que dos trayectos intermedios sortean este problema. La distancia se mide en milímetros.

```
float distancia() {                              (1)
 int tamanio = frames.size()-1;
 distancia = round(PVector.dist(getPosition(0),
                                  getPosition(tamanio/2)));
 distancia +=  round(PVector.dist(getPosition(tamanio/2),
                                  getPosition(tamanio)));

 return (distancia);
}

float velocidad() {                              (2)
  velocidad = distancia/(tiempo/1000);

  return velocidad;
}

float aceleracion() {                            (3)
  aceleracion = distancia/sq(tiempo/1000);

  return aceleracion;
}
```

Código 4.3.3. Métodos auxiliares.

Velocidad() (2) y *aceleración()* (3) completarán los valores cinemáticos que faltan. El código es sencillo y se basa en dos formulas. Para la velocidad se divide la distancia entre el tiempo (medido en segundos) y para la aceleración la distancia entre el tiempo al cuadrado (véase formulas 2.2).

```
  void recordFrame() {
    PVector position = new PVector();
    float confidence =
    kinect.getJointPositionSkeleton(userID, jointID, position);

    if (confidence > 0.5)
      frames.add(position);
  }

  void nextFrame() {
    currentFrame++;
    if (currentFrame == frames.size()) {
      currentFrame = 0;
    }
  }

  void clearFrames() {
    frames = new ArrayList();
  }
```

Código 4.3.4. Métodos auxiliares.

Las funciones que manejan los *frames* o posiciones de datos son *recordFrame()* que guarda en el array de vectores las posiciones por donde pasa el punto de control, *nextFrame()* que aumenta el contador de *frames* y *clearFrames()* que limpia el array. Siempre y cuando el valor *confidence* sea mayor a 0.5. Este valor se comento en capítulos anteriores (Cap. 2). El resto de métodos a excepción del que compila la salida de datos en un objeto JSON son métodos *get()* y *set()* necesarios para introducir o sacar los valores de las variables con las que trabaja esta clase.

4.4. Clase Pipe

Una de las ventajas de la utilización de múltiples hilos de ejecución en una aplicación, o aplicación *multithreaded*, es que pueden comunicarse entre sí. Se pueden diseñar hilos para utilizar objetos comunes, que cada hilo puede manipular independientemente de los otros hilos de ejecución. Así la clase *pipe* o tubería en castellano será el objeto encargado de almacenar los datos que se pasen los hilos. La clase *pipe* actuara como contenedor donde los hilos van dejando los valores, mediante *lanzar(),* calculados en un tiempo. El servidor dentro de un bucle (Capitulo 5) ira recogiendo mediante el método *recoger()* estos datos. Los datos tendrán un modelo concreto basado en JSON

4.4.1. Diagrama de flujo

Los dos métodos siguen los mismos pasos. A la hora de recoger o enviar esperan que el buffer este lleno o vacío respectivamente. Una vez comprobado el estado modifican las variables de control que regulan el buffer y recogen o guardan un nuevo paquete de datos. Siempre se tiene que notificar los movimientos de entrada-salida ya que tanto los hilos como el servidor trabajan en paralelo y podrían provocar colisiones o solapes a la hora de manejar el buffer.

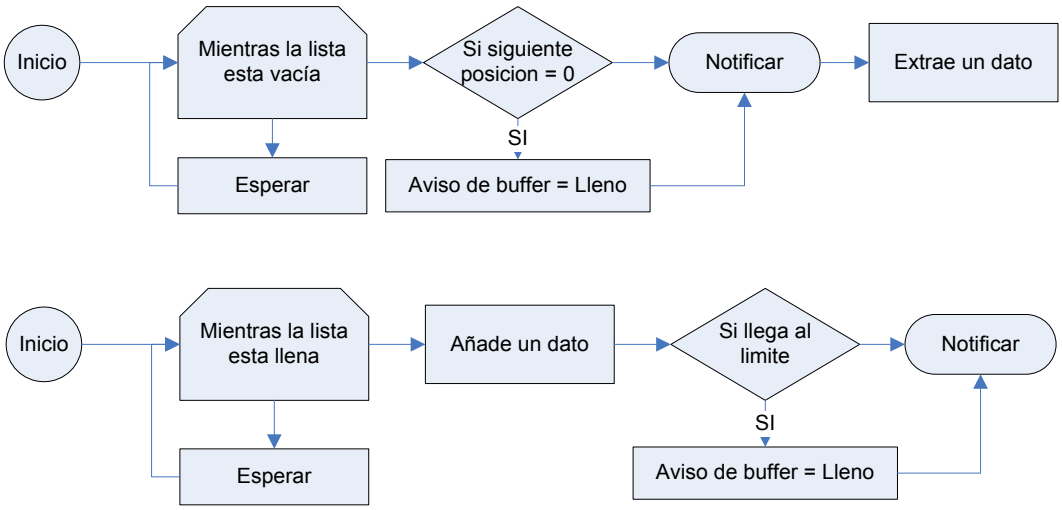

Diagrama 4.4. Flujo de los dos métodos de Pipe. Arriba Recoger y abajo Lanzar.

4.4.2. Codificación

Se muestra primero las variables que usadas y a continuación el código de los métodos principales.

```
final int NDATOS = 15;

class Pipe {

  private JSONObject buffer[] = new JSONObject[NDATOS];
  private int siguiente = 0;

  private boolean estaLlena = false;
  private boolean estaVacia = true;
```

Código 4.4. Variables.

En esta clase se trabajara mediante un buffer de objetos JSON (capitulo 5) y las variables booleanas *estaLlena* y *estaVacia.* Por último el iterador siguiente marcara en qué lugar del buffer se guardara o extraerá el objeto. No olvidar la constante *NDATOS* que contiene el tamaño del buffer.

```
public synchronized JSONObject recoger() {          (1)

   while( estaVacia == true ){
    try {                                           (2)
        wait();
    } catch( InterruptedException e ) {
        println("Recoger interrumpido");
    }
   }

   siguiente--;                                     (3)

   if( siguiente == 0 )
     estaVacia = true;                              (4)

   estaLlena = false;
     notify();                                      (5)

   return( buffer[siguiente] );
}
```

Código 4.4.1. Método recoger.

La función *recoger()* debe estar sincronizada ya que otros hilos van a tratar con ella a la vez. Es por eso que se instancia *synchronized* en la cabecera del método (1). Lo mismo pasara con *lanzar()*. Esto hace que los hilos sincronizados manden señales entre ellos para notificar el manejo de objetos comunes, en este caso los buffer.

Entrando en el código de *recoger()* la primera sentencia es un bucle que se realiza siempre y cuando el buffer está vacío. En ese caso se realiza una llamada a *wait()* quedando el hilo servidor (el único que recoge) en espera (2). Cuando se notifique un cambio en el buffer (un hilo a introducido sus datos) el servidor saldrá de la espera y significara que la variable de control ha cambiado. Cuando eso suceda al iterador (que siempre ira un valor por encima) se le resta uno (3). Si el valor del iterador es 0 significa que se va a extraer el último dato del buffer por tanto la variable *estaVacia* pasara a *true* (4). Obviamente ser marca *estaLlena* a falso ya que se extrae un objeto. Con el iterador en la posición correcta y las variables marcando el estado del buffer se notifica a los hilos y se extrae el objeto (5).

Lanzar() a diferencia de *recoger()* recibe un parámetro (1) desde hilos que será el objeto que debe ser guardado en la tubería a la espera del servidor. Por lo demás el código es semejante al anterior. Si el buffer está lleno habrá que esperar que el servidor

vacié el contenido (2). Mediante la llamada *wait()* se mantiene a los hilos esperando. Una vez haya hueco se guarda el dato en el buffer (3) y se incrementa el iterador. Si el valor de este coincide con la constante que marca el tamaño del buffer significara que está lleno. (4) Por tanto se marcara como lleno *estaLleno = true* y se desmarcara *estaVacia.* Se acaba con la notificación a los hilos que estén esperando, en este caso al *Server* de que ya hay datos para extraer (5).

```
public synchronized void lanzar( JSONObject s ) {      (1)

   while( estaLlena == true ){
       try {
          wait();                                      (2)
       } catch( InterruptedException e ) {
          println("Lanzar interrumpido");
       }
   }

   buffer[siguiente] = s;                              (3)

   siguiente++;

   if( siguiente == NDATOS )                           (4)
       estaLlena = true;
   estaVacia = false;
   notify();                                           (5)
}
```

Código 4.4.2. Método lanzar.

Capitulo 5. Estructura Cliente – Servidor

5.1. Introducción

A la hora de aplicar una solución respecto al problema referido con la gestión y análisis de los datos proporcionados por este programa se eligió implementar una arquitectura basada en un servidor y un cliente por varios motivos. De esta manera es más sencillo diferenciar entre las tareas de obtención y análisis de datos, haciendo que sea un proyecto escalable y consiguiendo dos módulos cliente y servidor. Además se aprovecha las librerías que se proporcionan, en este caso *Processing.net*, pero podría ser las propias funciones *socket* de Java.

Resumiendo el funcionamiento de la arquitectura C/S, el servidor gestiona los tiempos de envió hacia el cliente, una vez tenga un conjunto de datos (formado por los quince puntos de control) se enviara al cliente. La aplicación cliente que se muestra en esta memoria es una aplicación que se ha utilizado para realizar pruebas, no se pretende que sirva de resultado final ya que el análisis de datos no es objetivo de este proyecto. (Más información sobre C/S capitulo 2.6).

5.2. Modelo de datos de salida

5.2.1. Conceptos básicos sobre JSON

JSON acrónimo de *JavaScript Object Notation*, es un formato ligero para el intercambio de datos. Fácil de leer y escribir para los usuarios y simple de interpretar y generar par las maquinas. La simplicidad de este modelo ha dado lugar al aumento de su uso dejando a un lado otros modelos como XML. Una de las ventajas de JSON es que es mucho más sencillo escribir un analizador de datos o *parser*.

Está constituido por dos estructuras: una colección de pares nombre/valor, conocido también como objeto y una lista ordenada de valores, secuencia, lista, etc. Estas son estructuras universales para todos los lenguajes de programación algo razonable en un formato de intercambio de datos independiente del lenguaje.

5.2.1.1. Elementos de la estructura

Apoyándose en la estructura (Cód. 5) que valida la salida de datos del programa se comentan algunos elementos de JSON.

Objeto: Conjunto de datos desordenado de pares nombre/valor. Un objeto comienza con { y termina }. Cada nombre es seguido por : (dos puntos) y el par nombre/valor separado por , (coma).

Array: colección de valores, comienza con [y termina con]. Los valores se separan por , (coma).

Valor: un valor puede ser una cadena de caracteres con comillas dobles o un número, valor booleano incluso una array. Por ejemplo el valor del objeto *partes* es una array.

5.2.2. Plantilla JSON de salida

El archivo JSON que se muestra a continuación valida la salida de datos del programa.

```
{
   "datos": {
      "hora": "hora:min:sec.milis",                    Objeto      (1)
      "partes": [                                      Array
         {                                                        (2)
            "joint": "nombre",
            "id": "numero",
            "distancia": "valor",
            "aceleracion": " valor ",
            "velocidad": " valor"
         }
         {                                                        (2)
            "joint": " nombre ",
            "id": " numero ",
            "distancia": " valor ",
            "aceleracion": " valor ",
            "velocidad": " valor"
         }
         …
      ]
   }
}
```

Código 5. Estructura que siguen los datos de salida.

El primer objeto (*datos)* del archivo JSON se introducen en la clase Servidor. *Datos* tiene como valor otros objetos: *Hora*, el momento en el que se calculo los valores con formato hh:mm:ss,ms; y *partes*, una array con los resultados del punto de control. Este array se crea dentro de la clase *Hilo*. Tendremos quince elementos en el array. Los objetos que componen cada elemento son: *Distancia*, *tiempo* (desde que comenzó el programa), *aceleración*, *velocidad* y *joint* cuyo valor es un número que identifica a cada punto y servirá a la hora de ordenar la array en el servidor.

5.2.3. Métodos y tipos de datos

El formato de datos de salida empieza a componerse dentro de los hilos. Los valores de cada hilo se envían en un objeto JSON, uno por cada punto de control, hacia el servidor. El servidor una vez tiene los 15 objetos en el buffer de formar ordenada crea la salida respetando la estructura anterior. Los métodos que se usan en cada clase son *crearJSON()* y *empaquetarDatos().*

```
JSONObject crearJSON (float d, float v, float a) {
  JSONObject parametro = new JSONObject();

  parametro.setString("joint", this.getName());
  parametro.setInt("id", jointID);
  parametro.setFloat("distancia", distancia/10);
  parametro.setFloat("velocidad", velocidad/10);
  parametro.setFloat("aceleracion", aceleracion/10);

  return (parametro);
}
```

Código 5.1. Método crearJSON dentro de la clase Hilo.

El código anterior corresponde al método auxiliar *crearJSON()* que se encuentra dentro de la clase Hilo. Se pasan tres parámetros de tipo *float* distancia, velocidad y aceleración. Los objetos JSON de Processing tienen funciones para añadir y extraer *set()* y *get()* elementos que permiten formar el archivo jerarquizado. Gracias a un objeto JSON auxiliar *parámetro* se va compilando con cada uno de los elementos. Processing proporciona métodos dependiendo de los tipos de datos que se vayan a incluir dentro del objeto. La clase Hilo recibe el resultado del método, un objeto JSON con el formato correspondiente a los elementos de la array (Código 5 (2)).

```
JSONObject empaquetarDatos () {

   JSONObject json = new JSONObject();
   json.setString("hora",
        (hour()+":"+minute()+":"+second()+"."+millis()));

   JSONArray valores = new JSONArray();

   for (int i=0; i<NPARTES; i++) {                (1)
     valores.setJSONObject(i, buffer[i]);
   }
   json.setJSONArray("partes", valores);

   return(json);
 }
```

Código 5.2. Método empaquetarDatos() dentro de la clase Servidor.

La clase *servidor* forma el objeto final que se enviara al programa cliente. Este objeto ya corresponde con la plantilla de control (Cód. 5). Se crea un objeto JSON donde se añaden dos elementos, correspondientes con la cabecera (Cód. 5 (1)). *Hora* y el array. Con ayuda de un bucle *for* (1) se introducen todos los elementos del array de formar ordenada. Una vez se tiene el array completo se añade al objeto JSON mediante el método *setJSONArray().* El resultado es el objeto ordenado y conforme al modelo.

5.3. Clase Servidor

El proyecto ha sido diseñado para que una vez los hilos hayan obtenido las medidas necesarias un cliente, otro programa independiente, trabaje con los valores. Para ello se ha programado una clase *servidor* que extienda de *thread* y se mantenga enviando datos durante todo el proceso.

5.3.1. Diagrama de flujo

El diagrama de flujo de la clase *servidor* se basa en un bucle que mantendrá al servidor activo mientras no se reciba una señal de stop por parte del programa padre, o principal. Existen funciones auxiliares que ordenan, completan el objeto JSON de salida y regulan el envío siempre y cuando alguna parte del cuerpo se haya movido.

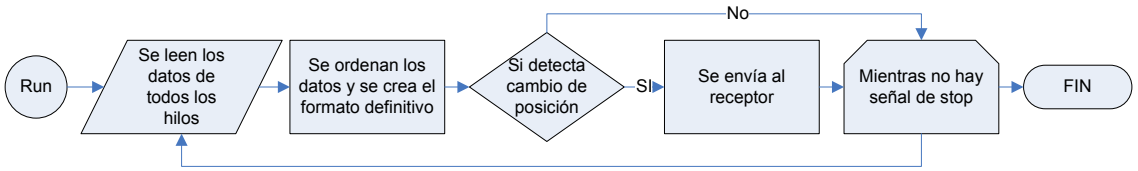

Diagrama 5.2. Flujo de la clase Servidor.

5.3.2. Codificación

Como siempre lo primero que aparece en el código son las bibliotecas importadas.

```
import java.lang.Object.*;
import java.util.*;

final int NPARTES = 15;

class Servidor extends Thread {

  private JSONObject buffer[] = new JSONObject[NPARTES];
  private Pipe p;
  private Server miServer;
  private Boolean stop = false;
  private Boolean envioOk = false;

  public Servidor (Pipe pipe, Server s) {    (1)
    p = pipe;
    miServer = s;
  }
```

Código 5.3. Codificación Servidor.

La clase como se ha dicho antes extiende de *thread* por tanto es un hilo y su ejecución dependerá de un padre que lo inicie mediante *start()*. A continuación se crea una constante de tipo *int* que marca el tamaño del buffer de entrada. Como variables dentro de la clase se crea el *buffer* de datos de entrada que será una array de tipo *JSONObject*. También se trabaja con un objeto del tipo *Pipe* para recibir los datos y un objeto *Server* que implementa el servidor de la biblioteca Network de Processing, además se tiene una variable booleana con el valor de la señal de stop que regula la salida del programa y por ultimo otra variable del mismo tipo que sirve de *flag* o marcador para efectuar el envío de datos.

El constructor (5.3 (1)) recibe dos parámetros. Un objeto *Pipe* que será la tubería de entrada y el servidor. Estos dos objetos se lo pasa el programa principal a la hora de crear el servidor y están inicializados. El servidor trabaja con ellos tal y como se pasan.

```
public void run() {
    JSONObject datoSalida = new JSONObject();

    println("Inicio el servidor");

    do {                                          (1)
      for (int i=0; i<NPARTES; i++) {
        buffer[i] = p.recoger();
        comprobar(buffer[i]);                     (2)
      }
      try {
        sleep(250);
      }
      catch(InterruptedException e) {
        ;
      }
      ordenarDatosEntrada();                      (3)
      datoSalida = empaquetarDatos();
      enviarDatos(datoSalida);
    }
    while (!stop);
  }
```

Código 5.3.1. Método *run()* dentro del Servidor.

Como cualquier hilo en Java el grueso de la codificación se encuentra en el método *run()*. Se tiene una variable *datoSalida* de tipo *JSONObject*. Dentro del bucle (1) que mantiene con vida al hilo se encuentra una primera sentencia *for* que mediante la tubería o *pipe* recoge los resultados que producen los hilos. Cada resultado es valorado por el método auxiliar *comprobar* (2). En el siguiente punto se explica su trabajo. Cada vez que se reciben todos los datos de un intervalo de tiempo el hilo duerme durante unos milisegundos, esto como ya se hizo con los *hilos* es para que las reiteraciones del bucle sean más lentas y no sobrecarguen el sistema. Con todos los datos en el *buffer* se ordenan (3) ya que al trabajar los quince hilos de forma paralela pueden llegar de forma aleatoria. Los datos de salida tienen que cumplir una forma concreta así que tras ordenarlos respecto a esta se completa el objeto JSON. Por último solo queda enviar los datos al cliente, que será un programa independiente conectado mediante los mecanismos de la librería Network de Processing quien analice y trabaje con los valores.

5.3.2.1. Métodos auxiliares

Se codifican varias funciones auxiliares. *OrdenarDatosEntrada()* (5.3.2) contiene dos bucles *for* anidados los cuales recorren el buffer buscando el valor id de cada punto de control y ordenándolos de menor a mayor

```
void ordenarDatosEntrada() {
    JSONObject aux = new JSONObject();

    for (int i = 0; i < buffer.length - 1; i++) {
      for (int j = i + 1; j < buffer.length; j++) {
        if (buffer[j].getInt("id") <
                                buffer[i].getInt("id")) {
          aux = buffer[i];
          buffer[i] = buffer[j];
          buffer[j] = aux;
        }
      }
    }
  }
```

Código 5.3.2. Método ordenarDatosSalida.

Las dos siguientes funciones son las encargadas del trabajo principal de esta clase *server*. Se trata de la comprobación de movimiento, es decir, siempre que llega un paquete de datos con los resultados de un punto del cuerpo se comprueba si la distancia es mayor que 0. Eso asegurara que esa parte del cuerpo ha sufrido movimiento. Si no hay movimiento en al menos un punto del usuario no se enviara nada al cliente.

La segunda función es *enviarDatos()*. Recibe un parámetro de tipo *JSONObject* en su cabecera. Por limitaciones de la librería Network de Processing este objeto se convierte en una cadena de caracteres antes de ser enviado al cliente (1). Esto no afecta a la sintaxis ni a la estructura del objeto de salida. Por parte del programa que implementa el cliente se puede trabajar con el objeto de salida sin problema. Se usa la variable *envioOK* para marcar si hay movimiento.

```
void comprobar(JSONObject obj){
   if (obj.getFloat("distancia") > 0)
     envioOk = true;
}
void enviarDatos(JSONObject json) {
   String sJson = json.toString();          (1)

   if(envioOk == true)
      miServer.write(sJson);
   envioOK = false;
}
```

Código 5.2.3. Método auxiliar en Servidor.

5.4. Clase Cliente

Aunque no es objetivo de este proyecto programar una aplicación que analice los datos se mostrara la codificación de una clase realizada sobre Processing que ha servido para hacer pruebas y depurar el servidor. Esta clase recibe los datos del servidor de forma remota y los exporta a un archivo con extensión *json*.

5.4.1. Codificación

Igual que en la clase servidor se ha de importar la librería de Processing Network. El objeto más importante de esta clase será el cliente, además se crean dos variables para la salida de datos: *dataString* variable de tipo *String* y *output* de tipo *PrintWriter*. Este tipo de objetos los proporciona Processing y sirven para sacar datos en formato texto.

Dentro de *setup()* (1) se crea la conexión con el servidor. El constructor de *client* nos pide tres valores. Un programa padre de tipo *PApplet* en este caso se usara *this* para referirse a la misma clase Cliente. El siguiente parámetro es la IP del servidor y por último el puerto por donde se trasmite. En el caso del puerto tiene que ser similar en cliente y servidor. Además de crear el cliente debemos de crear el archivo de salida donde se volcaran los datos. Entre paréntesis se introduce el nombre del archivo.

El método *draw()* (2) es el encargado de recibir los datos del servidor. Mediante la función *client.available()* se conoce si hay datos que recibir. Si eso es así se guardan en una variable auxiliar. La siguiente sentencia *if* sirve para comprobar si los datos recibidos son validos en el caso positivo se escriben en la salida. Como se puede

comprobar existen dos usos de *println()* el primero es para escribir sobre el archivo de salida y el segundo sobre la pantalla. Por último se reinicia la variable auxiliar a la espera del siguiente envío.

```
import processing.net.*;

Client miCliente;

String dataString;
PrintWriter output;

void setup() {                                                    (1)
  miCliente = new Client(this, "192.168.1.5", 5204);
  output = createWriter("Salida.JSON");
}

void draw() {                                                     (2)
  if (miCliente.available() > 0) {
    dataString = miCliente.readString();
  }
  if (dataString != null) {
    output.println(dataString);
    println(dataString);
    dataString = null;
  }
}

void keyPressed() {
  output.flush();
  output.close();
  exit();
}
```

Código 5.3. Clase Cliente

El cliente se desconectara en dos casos, si el servidor termina su ejecución o si el propio cliente termina. En el caso de terminar el cliente se cierran los archivos de escritura, si fuera el servidor que se cierra de forma abrupta se perderá los datos. Apuntar que en los casos probados siempre se ha cerrado el cliente antes que el servidor, y como se ha mencionado antes no es objetivo crear un cliente por tanto la opción de una desconexión remota por parte del servidor tendrá que tenerse en cuenta en futuras mejoras por parte de los encargados de este modulo.

Capitulo 6. Conclusiones y trabajos futuros

6.1. Conclusiones finales

En el trascurso de este proyecto se han estudiado herramientas y técnicas de tratamiento de imágenes, desarrollo de interfaces, herramientas de concurrencia, conocimientos de programación orientada a objetos en Java y empleando técnicas de análisis y diseño de ingeniera de *software* con el propósito de realizar un programa para la monitorización y gestión de un sistema de reconocimiento y calculo de medidas sobre un individuo. A día de hoy el abaratamiento de la tecnología y la liberación del código propietario, además del aumento de investigadores y proyectos de *software* libre permiten el desarrollo de proyectos como este.

Las facilidades que ofrece Processing como entorno de desarrollo libre basado en Java son muchas. Con un conocimiento medio de programación orientada a objetos y apoyándose en el API que ofrece la web oficial se pueden crear aplicaciones *software* complejas de forma sencilla. Processing cuenta con multitud de librerías que permiten interactuar con dispositivos *hardware* como Kinect o Arduino entre otros. Esto aumenta las posibilidades a la hora de realizar proyectos *software*. A la hora de comenzar a trabajar con este entorno se puede encontrar fácilmente documentación tanto en forma de libros como en páginas personales de desarrolladores.

Kinect brinda un abanico amplísimo de posibilidades a la hora de realizar programas *software* de todo tipo con un precio asequible. Como cámara para el mundo de los videojuegos es innegable su potencial aparte de eso sorprenden los proyectos que se han desarrollado y que se están desarrollando actualmente. Las características *software* de reconocimiento y diferenciación de usuarios basadas en sus recursos *hardware* están creando interés hasta el punto que otras marcas comerciales han desarrollado sus propias cámaras de profundidad fuera del mundo del ocio con vistas a la investigación.

6.1.1. Valoración de resultados

Tras la realización de este proyecto los objetivos que se marcaron al inicio han sido cumplidos con buen resultado. La mayor dificultad ha residido a la hora de conseguir resultados realistas en las medidas. No es trivial comprobar de forma manual medidas en tres dimensiones ya que se está acostumbrado a trabajar en planos y no existen medidores de velocidad o aceleración puntual que se puedan usar en las pruebas. El proceso de pruebas ha sido de lo más costoso de este proyecto. Se ha tenido que modificar en varias ocasiones los algoritmos de cálculo de distancias de manera que la distorsión que introduce la cámara en las posiciones puntuales es constante. El resultado ha sido medidas con una precisión optima que permite unos resultados finales mejores de los que se esperaba al inicio.

Sobre la base inicial del proyecto se ha añadido de poder hacer que la aplicación se comunique mediante un servidor con otra remota que se implementara en sucesivos proyectos. Creemos que es una mejora necesaria para futuros proyectos basados en este.

6.1.2. Márgenes de error

No es despreciable el ruido o distorsión en las medidas que introduce Kinect esto depende de las condiciones en el proceso de calibrado. Como se ha dicho al usar cámara IR la luz no influye en el proceso pero ciertos objetos pueden crear reflexiones y en consecuencias zonas oscuras donde la imagen no tiene valores conocidos. Es un efecto parecido a las zonas de sombra donde desparecen las referencias de las partes del cuerpo pero a diferencia de esto con las reflexiones no se puede solucionar de manera sencilla.

6.2. Mejoras y trabajos futuros

6.2.1. Seguimiento múltiple

En los primeros capítulos donde se habla sobre la calibración y las posibilidades de SimpleOpenNi se comenta la capacidad para reconocer y seguir varios usuarios. Una futura mejora o ampliación de este proyecto sería calcular los valores para múltiples personas. La solución es sencilla, crear otro grupo de hilos para los diferentes *ids* que devuelve la calibración. La limitación estaría en la capacidad de la CPU usada, ya que se aumentaría considerablemente el número de procesos corriendo en paralelo.

Imagen 19. Varios usuarios diferenciados por su ID.

6.2.2. Triangulación

Para solucionar los problemas que se han mencionado antes existe la posibilidad de triangular datos con múltiples cámaras enfocadas en puntos equidistantes hacía un mismo objetivo. Este sería el escenario ideal, varios puntos de referencia, tener datos desde tres ángulos diferentes para no perder medidas por sombras y reflexiones. La solución es muy parecida al famoso ojo de halcón que se implemento en el tenis profesional. Aunque pueda parecer sencillo hay que estudiar los ángulos y las distancias a los objetos antes de colocar las cámaras además del coste en material que aumentaría por cada cámara.

No se han realizado pruebas con varios individuos en este proyecto, sería interesante tratar el momento en el que los usuarios se colocarán de manera que uno de ellos quedara tapado por los demás. En esos casos las posibilidades de que un usuario entrara en sombra es claramente más alto que en el caso actual. La triangulación de cámaras vuelve a salir como solución posible a las sombras en caso de varias personas en pantalla.

6.2.3. Captura de movimientos y robótica

Actualmente es corriente que las compañías de animación usen técnicas de captura de movimiento mediante sensores adosados en el cuerpo de actores. Con esta

técnica se crea un modelo en 3D de movimiento aplicable a videojuegos, películas de animación etc. Kinect mediante un *software* especifico y una triangulación estudiada de cámaras ofrece la posibilidad de calcular el volumen y capturar los movimientos del individuo. No es una novedad en este aspecto si nos referimos a grandes compañías pero si desde un punto de vista para pequeños desarrolladores gracias al precio y facilidad de recursos que se pueden encontrar. Si sumamos estos modelos de movimiento con robótica y electrónica apoyada en *hardware* libre como Arduino se pueden crear proyectos ambiciosos de bajo coste.

Imagen 20. Proyecto Kinect más robótica.

Presupuesto

A continuación se detalla el coste del material utilizado y las horas de trabajo.

Material utilizado

	Precio/Unidad	**Unidades**	**Total**
Ordenador portátil	600€	1	600€
Kinect	149€	1	149€
		Total Material:	749€

Horas de trabajo

	Horas	**Precio/Horas**	**Total**
Análisis previo	40	30	1200€
Estudio y diseño del método	80	30	2400€
Codificación	120	30	3600€
Pruebas	80	30	2400€
Total horas:	360		9.600€

Total proyecto: 10.349€

Tabla 2. Presupuesto.

Bibliografía

Libros de texto y digitales:

REAS, Casey, FRY, Ben. *Processing a programming handbook for visual Designers and artists.* Cambridge, MA, USA: MIT Press, 2012. ISBN: 978-0-262-18262-1.

SHIFFMAN, Daniel. *Learning Processing*. Burlington, MA, USA: Morgan Kaufmann, 2008. ISBN: 978-0-12-373602-4.

BORENSTEIN, Greg. *Making things see.* 1ª ed. Canada: O'Reilly Media, 2012. ISBN: 978-1-449-30707-3.

REAS, Casey, FRY, Ben. *Getting started with Processing.* 1ª ed. Canada: O´Reilly Media, 2010. ISBN: 978-1-449-37980-3.

KEAN, Sean, HALL, Jonathan y PERRY Phoenix. *Meet the Kinect*. USA: Apress, 2011. ISBN: 978-1-4302-3888-1.

ECKEL, Bruce. *Piensa en Java.* Javier Parra (Rev. Tec.). España: Pearson Education, 2002. 906 p. ISBN: 978-8-4205-3192-2.

GOLDSTEIN, H. *Mecánica clásica.* Dr. Julián Fernández Ferrer (trad.) 2ª ed. España: Editorial Reverte S.A, 2006. ISBN 84-291-4306-8.

Recursos Web:

FRY, B., REAS, C. *Processing 2.* [En línea].
<http://processing.org/> [Consultado: todo el proceso].

GOOGLE PROJECT. *OpenNi libray for Processing.* [En línea].
<https://code.google.com/p/simple-openni/> [Consultado: 03-12-2012].

PRIMESENSE, LTD. *OpenSource SDK for 3D sensors.* [En línea].
<http://www.openni.org/> [Consultado: 03-12-2012].

INFAIMON. *Sistema de visión artificial.* [En línea].
<http://blog.infaimon.com/> [Consultado: 13-01-2013].

SHIFFMAN, D. *Daniel Shiffman.* [En línea].
<http://shiffman.net/p5/kinect/> [Consultado: 15-03-2013].

COLECTIVO blablaLAB. *blablaLAB.* [En línea].
<http://www.blablablab.org/> [Consultado: 02-04-2013].

iFIXIT. *iFixit: free repair manual*. [En línea]
<http://www.ifixit.com> [Consultado: 21-06-2013].

GONZÁLEZ, I., SANCHEZ, A.J., VICENTE, D. *Java Threads.* [en línea]. Salamanca, España: Dep. Informática Universidad de Salamanca, 2002. [Consultado 08-02-2013]. Web. <http://zarza.usal.es/~fgarcia/docencia/poo/01-02/trabajos/S3T3.pdf>

JSON [En línea]
<http://www.json.org/json-es.html> [Consultado 12-04-2013].

Glosario

→ **xBox 360**: Plataforma de juegos creada y comercializada por Microsoft.

→ **Software**: Equipamiento lógico o soporte lógico de un sistema informático.

→ **Hardware**: Partes tangibles o físicas de un sistema informático.

→ **Biometría**: Método o estudio para el reconocimiento humano basado en rasgos físicos.

→ **RGB**: Modelo de color representado por tres colores (Rojo, Verde, Azul).

→ **GNU**: Dice ser de los proyectos de software o hardware de desarrollo libre.

→ **SDK**: Kit de desarrollo de software.

→ **Telediagnostico**: Diagnostico a distancia a través de medios telemáticos.

→ **SI**: Siglas de Sistema Internacional de unidades.

→ **ASUS**: Compañía taiwanesa de productos electrónicos e informáticos.

→ **IR**: Radiación infrarroja.

→ **Frame Rate (fps):** Fotogramas por segundo.

→ **LGPL**: Licencia Publica General para bibliotecas GNU. Licencia software que garantiza la libertad de compartir y modificar software cubierto por ella.

→ **IDE**: Entorno de desarrollo integrado.

→ **MIT**: Instituto tecnológico de Massachusetts.

→ **Java**: Lenguaje de programación orientado a objetos desarrollado por Sun Microsystems.

→ **API**: Interfaz de programación de aplicaciones, conjunto de funciones y procedimientos.

→ **OpenGL**: Especificación que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

→ **Eclipse**: Entorno de desarrollo de código abierto multiplataforma para lenguaje Java.

→ **NetBeans**: Entorno de desarrollo de código abierto multiplataforma para lenguaje Java.

→ **Stickman**: Modelo humano básico y basado en líneas que unen puntos del cuerpo.

→ **Bodytracking**: Seguimiento del cuerpo humano por medio de procesos informáticos.

→ **XML**: Lenguaje basado en etiquetas o marcas utilizado para almacenar información.

→ **Arduino**: Plataforma hardware de libre desarrollo basada en una placa con un microcontrolador.

→ **C/S**: Cliente-Servidor.

→ **LAN**: Local area network. Red de área local.

→ **WAN**: Wide area network. Red de área mundial.

Apéndice A. Instalación de librerías y entornos necesarios.

Antes de empezar a escribir código y trabajar con los datos que ofrece Kinect, es necesario instalar la biblioteca de Processing que se va a utilizar. Como ya se ha mencionado (capítulo 2.4) se va a utilizar una biblioteca llamada SimpleOpenNI. Esta biblioteca proporciona acceso a todos los datos de Kinect que se necesitan en este proyecto, así como una serie de herramientas y asistentes.

Los pasos para la instalación de una biblioteca en Processing varían ligeramente en función del sistema operativo. Este apéndice da las instrucciones para instalar SimpleOpenNI en Windows 7, al final de este documento se citara brevemente como instalar los componentes en Mac OS X y Linux. Estas instrucciones pueden cambiar con el tiempo, para más información, consulte la página de instalación SimpleOpenNI[20].

Instalación de SimpleOpenNI ocurre en dos fases. En primer lugar, se instala en sí OpenNI. Este es el sistema de software proporcionado por PrimeSense que se comunica con el Kinect para acceder y procesar sus datos. Los pasos a seguir en esta fase difieren en función de su sistema operativo

Una vez se tiene instalado OpenNI es turno de la instalación de la biblioteca. Este proceso es estándar para todos los sistemas operativos. La guía que se muestra a continuación para este paso está basada en Windows 7 pero es similar en todos los SSOO.

[20] Guía de instalación para todos los SSOO. http://code.google.com/p/simple-openni/wiki/Installation.

Repositorio

Open Ni y Simple OpenNi - Paquetes de instalación

Imagen A.1. Web donde se encuentra el repositorio[21].

El primer enlace corresponde a la biblioteca SimpleOpenNI compatible con todos los SSOO, el resto de enlaces corresponden a los distintos tipos de paquetes de instalación de PrimeSense según el SSOO.

[21] http://code.google.com/p/simple-openni/downloads/list [Consultado: Feb-2013]

Guía de instalación.

Comprobar el sistema

Antes de comenzar la instalación es necesario revisar dentro de *Panel de Control > Sistema y Seguridad > Administración de dispositivos* si el sistema reconoce la cámara. Si no se ha intentando una instalación anterior el sistema no debería de reconocer ningún dispositivo. La cámara no tiene que estar físicamente conectada al equipo durante la comprobación.

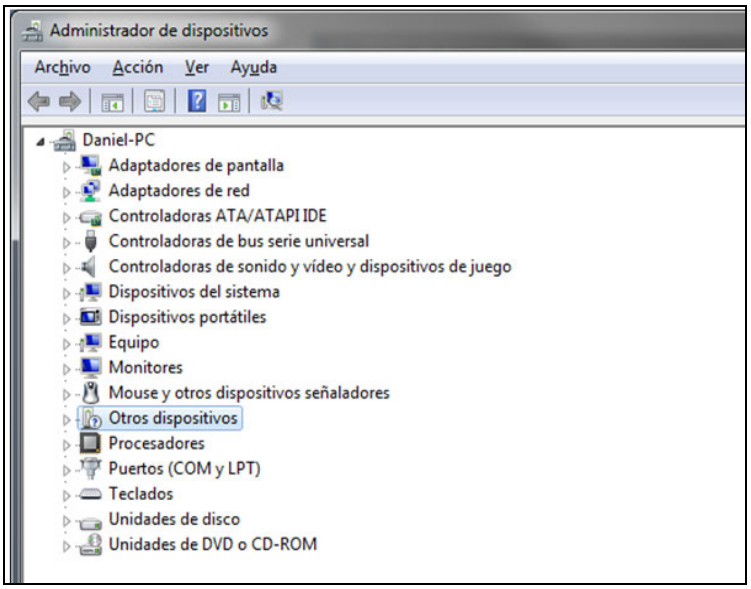

Imagen A.2. Comprobación del sistema en Windows7.

Instalar OpenNi

Instalar la plataforma Open Ni. Se ejecuta el archivo de instalación que se encuentra dentro del paquete de instalación.

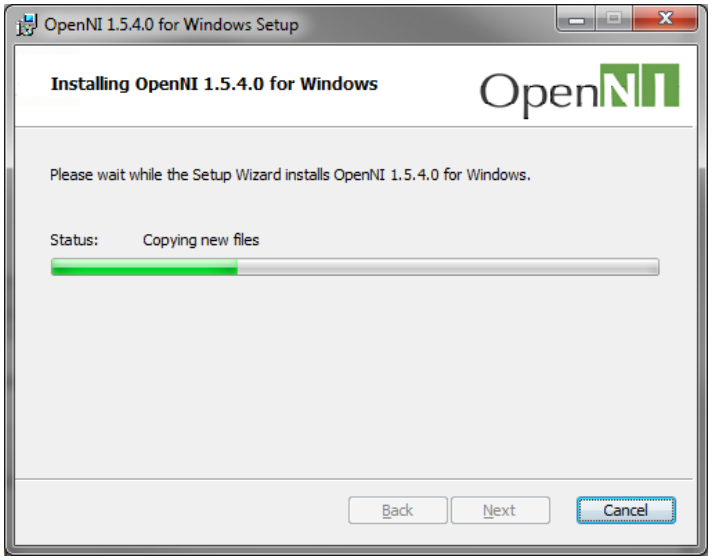

Imagen A.3. Instalación de OpenNi Windows7.

Una vez completada la instalación se crean una serie de directorios en el sistema donde se encuentra el archivo de instalación, una guía de usuario y varios ejemplos los cuales servirán de prueba una vez estén instalados los drivers.

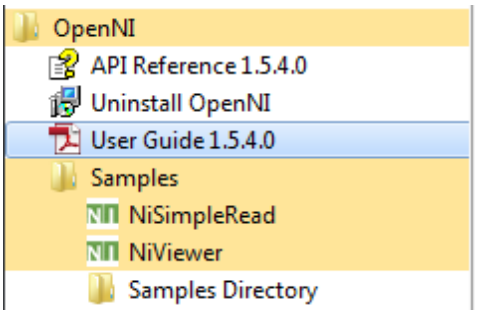

Imagen A.4. Archivos instalados Windows 7.

Instalación de NITE

Como en el paso anterior la instalación del componente NITE creara en el sistema una serie de directorios, documentación y ejemplos que permiten comprobar la funcionalidad de la cámara.

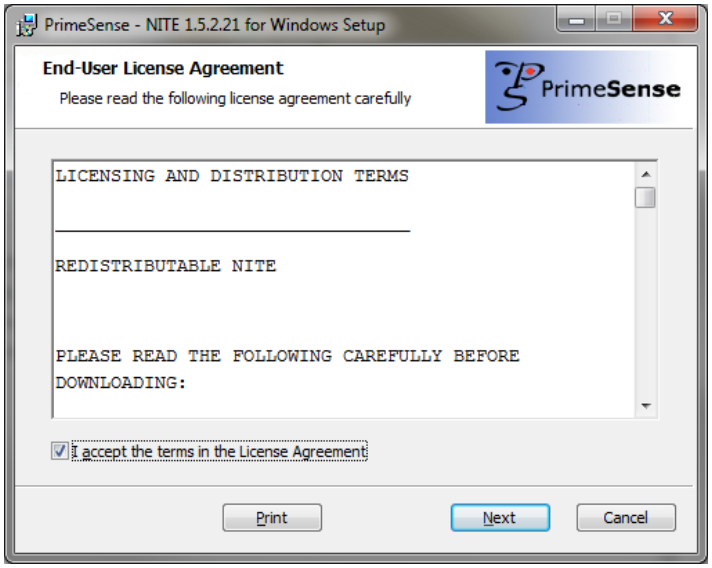

Imagen A.5. Instalación de PrimeSense Nite Win7.

Instalación Primesense Drivers

Instalación de los drivers de Primesense. Dentro del paquete de instalación el archivo a ejecutar tiene el nombre de *sensor-win32* en este caso. Dentro de la carpeta NITE se crea un directorio llamado sensor.

Instalación Kinect Drivers

Último punto de la instalación de los drivers y componentes. En este paso se necesita instalar los drivers de Kinect para que el sistema reconozca que el dispositivo está conectado.

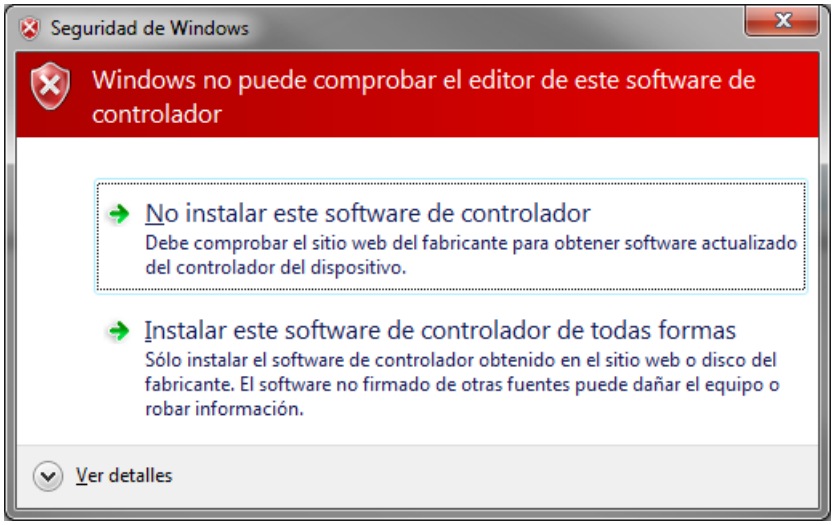

Imagen A.6. Mensaje de seguridad de Windows 7.

Antes de finalizar la instalación, depende de la configuración de seguridad configurada en el sistema, Windows avisa con un mensaje que la firma del controlador no es reconoce. Se aceptara el mensaje instalando el controlador y finalizara la instalación.

Comprobación de la instalación de la cámara

Tras el último paso se reinicia el sistema y se conecta físicamente la cámara al equipo. Una vez conectada la cámara deben salir unos mensajes avisando de un nuevo *hardware* encontrado. Para comprobar que todo a ha ido correctamente se vuelve a revisar en el Administrador de Dispositivos como indica el paso inicial.

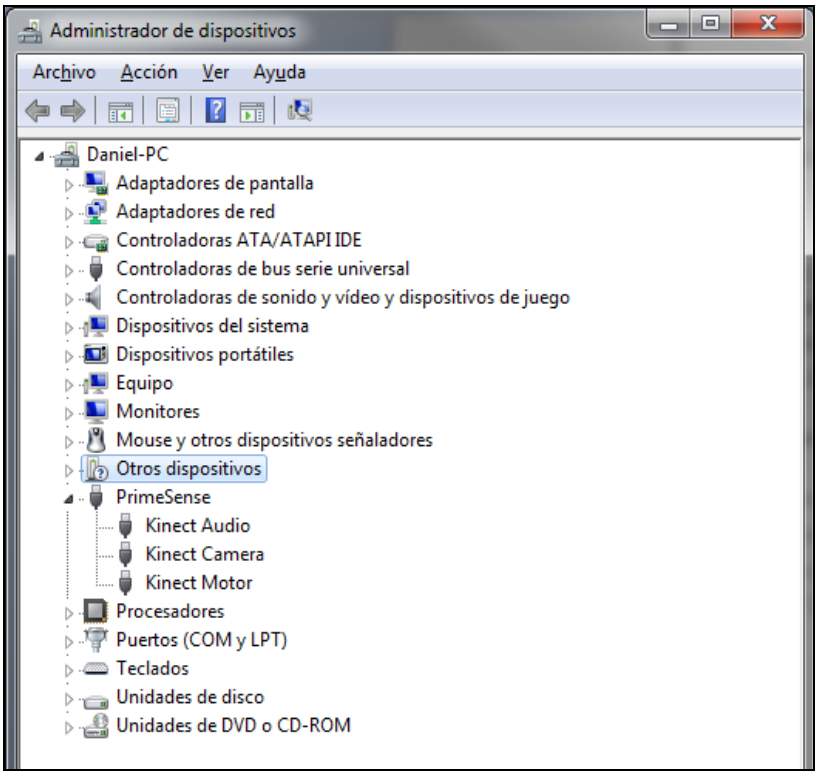

Imagen A.7. Comprobación final de instalación Win7.

Se puede comprobar que los controladores de PrimeSense se han instalado y se reconocen por el sistema. El led verde situado en el frontal de la cámara debe de parpadear confirmando que la cámara esta lista.

Instalación sobre OS X y Linux

Breve guía para instalar las librerías y componentes sobre los sistemas Mac OS X y Linux.

MAC OS X

a. Descargar el instalador del repositorio web (capítulo 1).
b. Descomprimir el archivo .zip en el directorio *OpenNI_NITE_Installer-OSX*
c. Abrir el terminal *Aplicaciones->Utilidades->Terminal*
d. Ir al directorio de instalación (Ilustración XXX)
e. Iniciar la instalación mediante el comando *sudo ./install.sh*. Escribir el password de administrador si fuera necesario.
f. Continuar con la instalación de Processing.

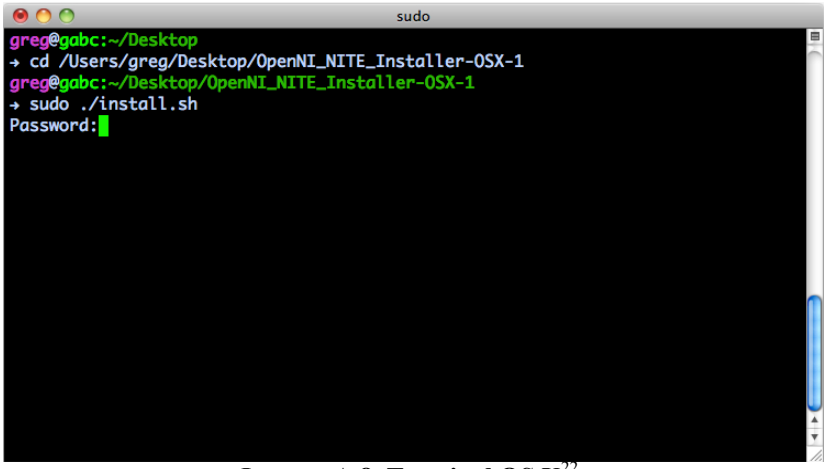

Imagen A.8. Terminal OS X[22].

LINUX

El proceso es relativamente similar al proceso de instalación de Windows excepto que existe la opción de descargar e instalar los binarios para todos los paquetes (OpenNI, NITE, y SensorKinect) o descargar y compilar el código fuente[23].

Instalar Processing

Processing es una plataforma que permite programar la interfaz grafica del proyecto. Para trabajar con ella en el sistema hace falta descomprimir el archivo descargado previamente. Se muestra la instalación sobre Windows 7 por ser similar en todas las plataformas.

[22] Ilustración obtenida de Making Things See – Greg Borenstein (2012)
[23] Más información en https://code.google.com/p/simple-openni/wiki/Installation#Linux

SimpleOpenNi sobre Processing

El último paso para tener completa la plataforma de desarrollo es incluir las librerías de SimpleOpenNi dentro de Processing. De esta manera a la hora de programar se podrá llamar a los métodos y funcionalidades que ofrece Kinect.

Es necesario copiar el archivo comprimido SimpleOpenNi dentro de la carpeta Libraries. La ruta es C:\Users\'nombre usuario'\Documents\Processing\libraries
Para comprobar si la librería ha sido exportada satisfactoriamente se observa dentro de File > Examples que existe un directorio de ejemplos titulado OpenNi.

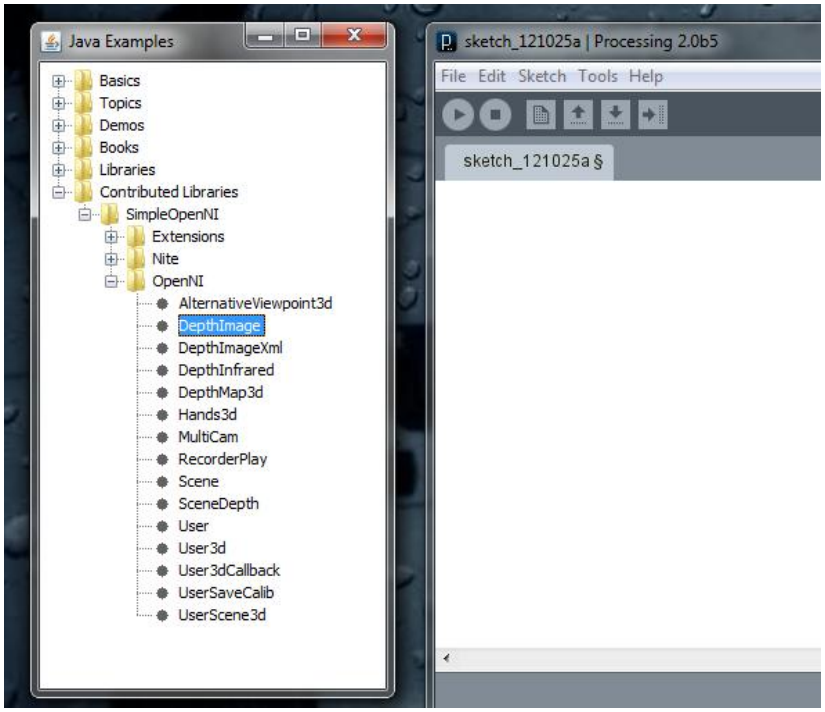

Imagen A.9. Processing con librerías instaladas Win7.

Conectar Kinect

Con las bibliotecas instaladas, se conecta la Kinect a un enchufe eléctrico, a continuación, conecte el extremo hembra del cable en forma de Y que viene del cable de alimentación como se muestra en la ilustración XXX. Esto debe hacer que se ilumine el indicador LED verde. El otro extremo de este cable en Y tiene un conector USB macho estándar. Se conectara al equipo como se muestra en la ilustración. Se debe poner la Kinect en el lugar conveniente, preferiblemente hacia donde se mira.

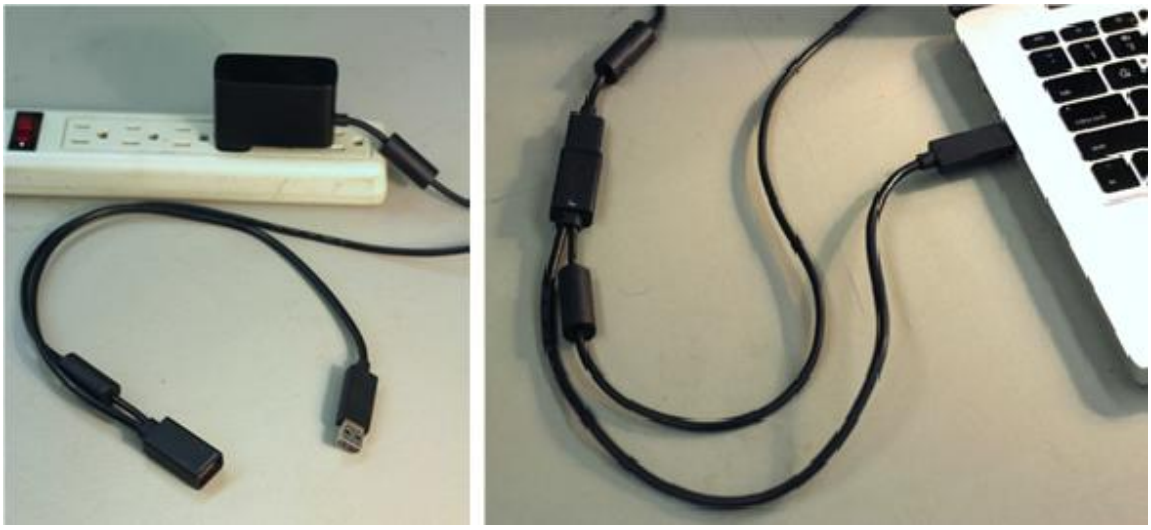

Imagen A.10. Cable eléctrico en forma de Y. Conector USB donde acaba el cable en Y enchufado al PC.

Imagen A.11. Kinect, conector macho que se conectara a la XBOX o en este caso al conector hembra que se encuentra en el cable en formar de Y[24].

[24] Imágenes punto 5 obtenidas de Making Things See – Greg Borenstein (2012)

ANEXO – Codificaciones

A. Código Skeleton.pde

En el capítulo 3 se explica el funcionamiento de la clase Skeleton. En resumen, es una clase auxiliar que implementa la parte gráfica del esqueleto del usuario. El programa principal en su método *draw()* hace una llamada a esta clase que dibuja los puntos del cuerpo del usuario si este está calibrado.

```
class Skeleton {

  SimpleOpenNI kinect;

  Skeleton(SimpleOpenNI context){
    kinect = context;
  }

  void drawSkeleton(int userId) {
    stroke(0);
    strokeWeight(5);

    kinect.drawLimb(userId, SimpleOpenNI.SKEL_HEAD,
    SimpleOpenNI.SKEL_NECK);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_NECK,
    SimpleOpenNI.SKEL_LEFT_SHOULDER);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
    SimpleOpenNI.SKEL_LEFT_ELBOW);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_ELBOW,
    SimpleOpenNI.SKEL_LEFT_HAND);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_NECK,
    SimpleOpenNI.SKEL_RIGHT_SHOULDER);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
    SimpleOpenNI.SKEL_RIGHT_ELBOW);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW,
    SimpleOpenNI.SKEL_RIGHT_HAND);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
    SimpleOpenNI.SKEL_TORSO);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
    SimpleOpenNI.SKEL_TORSO);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
    SimpleOpenNI.SKEL_LEFT_HIP);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_HIP,
    SimpleOpenNI.SKEL_LEFT_KNEE);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_KNEE,
    SimpleOpenNI.SKEL_LEFT_FOOT);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
    SimpleOpenNI.SKEL_RIGHT_HIP);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_HIP,
    SimpleOpenNI.SKEL_RIGHT_KNEE);
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_KNEE,
    SimpleOpenNI.SKEL_RIGHT_FOOT);
```

```
    kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_HIP,
    SimpleOpenNI.SKEL_LEFT_HIP);

    noStroke();
    fill(255, 0, 0);
    drawJoint(userId, SimpleOpenNI.SKEL_HEAD);
    drawJoint(userId, SimpleOpenNI.SKEL_NECK);
    drawJoint(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER);
    drawJoint(userId, SimpleOpenNI.SKEL_LEFT_ELBOW);
    drawJoint(userId, SimpleOpenNI.SKEL_NECK);
    drawJoint(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER);
    drawJoint(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW);
    drawJoint(userId, SimpleOpenNI.SKEL_TORSO);
    drawJoint(userId, SimpleOpenNI.SKEL_LEFT_HIP);
    drawJoint(userId, SimpleOpenNI.SKEL_LEFT_KNEE);
    drawJoint(userId, SimpleOpenNI.SKEL_RIGHT_HIP);
    drawJoint(userId, SimpleOpenNI.SKEL_LEFT_FOOT);
    drawJoint(userId, SimpleOpenNI.SKEL_RIGHT_KNEE);
    drawJoint(userId, SimpleOpenNI.SKEL_LEFT_HIP);
    drawJoint(userId, SimpleOpenNI.SKEL_RIGHT_FOOT);
    drawJoint(userId, SimpleOpenNI.SKEL_RIGHT_HAND);
    drawJoint(userId, SimpleOpenNI.SKEL_LEFT_HAND);
  }
  void drawJoint(int userId, int jointID) {
    PVector joint = new PVector();
    float confidence = kinect.getJointPositionSkeleton(userId,
jointID, joint);

    if (confidence < 0.7) {
      return;
    }

    PVector convertedJoint = new PVector();
    kinect.convertRealWorldToProjective(joint, convertedJoint);

    ellipse(convertedJoint.x, convertedJoint.y, 10, 10);
  }
}
```

B. Código Reloj.pde

Clase auxiliar para manejar los tiempos de la aplicación.

```
class Reloj {

  //inicia el contador
  float startTime() {
    float startTime = millis();
    return (startTime);
  }

  //devuelve el GAP de tiempo entre el inicio y ese momento ms
  float endTime(float startTime) {
    float elapsed = millis() - startTime;
    return (elapsed);
  }
}
```

C. Código Pipe.pde

```
final int NDATOS = 15;

class Pipe {

    private JSONObject buffer[] = new JSONObject[NDATOS];
    private int siguiente = 0;
    private boolean estaLlena = false;
    private boolean estaVacia = true;

    public synchronized JSONObject recoger() {
        while( estaVacia == true ){
            try {
                wait();
            } catch( InterruptedException e ) {
                println("Recoger interrumpido");
            }
        }

        siguiente--;

        if( siguiente == 0 )
            estaVacia = true;

        estaLlena = false;
        notify();

        return( buffer[siguiente] );
        }


    public synchronized void lanzar( JSONObject s ) {

        while( estaLlena == true ){
            try {
                wait();
            } catch( InterruptedException e ) {
                println("Lanzar interrumpido");
            }
        }

        buffer[siguiente] = s;

        siguiente++;

        if( siguiente == NDATOS )
            estaLlena = true;
        estaVacia = false;
        notify();
        }
    }
```

D. Código Principal.pde

```
import SimpleOpenNI.*;
import java.lang.Object.*;
import processing.net.*;

final int NHILOS = 15;
final int PUERTO = 5204;

SimpleOpenNI kinect;
Server server;
Servidor servidor;
Skeleton sk;
Reloj clock;
Pipe pipe;
Hilo[] aHilos = new Hilo[NHILOS];

/*VARIABLES*/
float startTime;
float tiempo=500;

void setup() {
  size(640, 480);
  frameRate(30);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();
  kinect.enableRGB();
  kinect.alternativeViewPointDepthToImage();
  kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);

  clock = new Reloj();
  pipe = new Pipe();
  sk = new Skeleton(kinect);

  server = new Server(this, PUERTO);
  servidor = new Servidor(pipe, server);
  servidor.start();

  crearHilos();
}
```

```
void draw() {
  background(0);
  kinect.update();
  kinect.setMirror(true);
  image(kinect.depthImage(), 0, 0);
  image(kinect.rgbImage(), 640, 0);

  IntVector userList = new IntVector();
  kinect.getUsers(userList);

  if (userList.size() > 0) {
    int userId = userList.get(0);
    insertarUsuario(userId);

    if (kinect.isTrackingSkeleton(userId)) {

      iniciarHilos();

      sk.drawSkeleton(userId);
      textSize(15);
      text("Segundos: " + clock.endTime(startTime)/1000 +" s",
                                                    100, 30);
    }
  }
}

void iniciarHilos() {
  for (int i=0; i<NHILOS; i++) {
    if (aHilos[i].getState() == Thread.State.NEW) {
      aHilos[i].setStartTime(startTime);
      aHilos[i].start();
    }
  }
}
void pararHilos() {
  for (int i=0; i<NHILOS; i++) {
    aHilos[i].pararHilo();
  }
}
void crearHilos() {
  String lines[] = loadStrings("BaseDatos.txt");
  for (int i=0; i<NHILOS; i++) {
    String words[] = lines[i].split(",");
    aHilos[i] = new Hilo(kinect, Integer.parseInt(words[0]),
tiempo, pipe);
    aHilos[i].setName(words[2]);
  }
}
void insertarUsuario(int userID) {
  for (int i=0; i<NHILOS; i++) {
    aHilos[i].setUser(userID);
  }
}
```

```
void keyPressed() {
  if (key == 'r') {
    startTime = clock.startTime();
    loop();
  }
  if (key == 'f') {
    noLoop();
  }
  if (key == ESC) {
    servidor.pararHilo();
    pararHilos();
  }
}
void onNewUser(int userId) {
  println("start pose detection");
  kinect.startPoseDetection("Psi", userId);
}
void onEndCalibration(int userId, boolean successful) {
  if (successful) {
    println(" User calibrated !!!");
    kinect.startTrackingSkeleton(userId);
    startTime = clock.startTime();
  }
  else {
    println(" Failed to calibrate user !!!");
    kinect.startPoseDetection("Psi", userId);
  }
}
void onStartPose(String pose, int userId) {
  println("Started pose for user");
  kinect.stopPoseDetection(userId);
  kinect.requestCalibrationSkeleton(userId, true);
}
```

E. Código Servidor.pde

```
import java.lang.Object.*;
import java.util.*;

final int NPARTES = 15;

class Servidor extends Thread {

private Server miServer;
private Boolean stop = false;
private JSONObject buffer[] = new JSONObject[NPARTES];
private Pipe p;
private Boolean envioOk = false;

  public Servidor (Pipe pipe, Server s) {
    p = pipe;
    miServer = s;
  }

  public void run() {
    JSONObject datoSalida = new JSONObject();

    println("Inicio el servidor");

    do {
      for (int i=0; i<NPARTES; i++) {
        buffer[i] = p.recoger();
        comprobar(buffer[i]);
      }
      try {
        sleep(250);
      }
      catch(InterruptedException e) {
        ;
      }

      ordenarDatosEntrada();
      //CREAR UN MODELO DE DATOS CONSTANTE
      datoSalida = empaquetarDatos();
      enviarDatos(datoSalida);
    }
    while (!stop);
  }
```

```
  void ordenarDatosEntrada() {
    JSONObject aux = new JSONObject();

    for (int i = 0; i < buffer.length - 1; i++) {
      for (int j = i + 1; j < buffer.length; j++) {
        if (buffer[j].getInt("id") < buffer[i].getInt("id")) {
          aux = buffer[i];
          buffer[i] = buffer[j];
          buffer[j] = aux;
        }
      }
    }
  }

  JSONObject empaquetarDatos () {

    JSONObject json = new JSONObject();
    json.setString("hora",
              (hour()+":"+minute()+":"+second()+"."+millis()));
    JSONArray valores = new JSONArray();
    for (int i=0; i<NPARTES; i++) {
      valores.setJSONObject(i, buffer[i]);
    }
    json.setJSONArray("partes", valores);

    return(json);
  }

  void comprobar(JSONObject obj){
   if (obj.getFloat("distancia") > 0)
     envioOk = true;
  }

  void enviarDatos(JSONObject json) {
    String sJson = json.toString();

    if(envioOk == true)
      miServer.write(sJson);

    envioOk = false;
  }

  void serverEvent(Server someServer, Client someClient) {
    println("Conexion entrante "+someClient.ip());
  }

  void pararHilo() {
    stop = true;
  }
}
```

F. Código Hilo.pde

```
import SimpleOpenNI.*;
import java.lang.Object.*;
import processing.net.*;

final int NHILOS = 15;
final int PUERTO = 5204;

SimpleOpenNI kinect;
Server server;
Servidor servidor;
Skeleton sk;
Reloj clock;
Pipe pipe;
Hilo[] aHilos = new Hilo[NHILOS];

/*VARIABLES*/
float startTime;
float tiempo=500;

void setup() {
  size(640, 480);
  frameRate(30);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();
  kinect.enableRGB();
  kinect.alternativeViewPointDepthToImage();
  kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);

  clock = new Reloj();
  pipe = new Pipe();
  sk = new Skeleton(kinect);

  server = new Server(this, PUERTO);
  servidor = new Servidor(pipe, server);
  servidor.start();

  crearHilos();
}

void draw() {
  background(0);
  kinect.update();
  kinect.setMirror(true);
  image(kinect.depthImage(), 0, 0);
   image(kinect.rgbImage(), 640, 0);
```

```
IntVector userList = new IntVector();
  kinect.getUsers(userList);

  if (userList.size() > 0) {
    int userId = userList.get(0);
    insertarUsuario(userId);

    if (kinect.isTrackingSkeleton(userId)) {
      //inicia los hilos
      iniciarHilos();
      //pinta el skeleton
      sk.drawSkeleton(userId);
      textSize(15);
      text("Segundos: " + clock.endTime(startTime)/1000 +" s"
                                                  , 100, 30);
    }
  }
}

void iniciarHilos() {
  for (int i=0; i<NHILOS; i++) {
    if (aHilos[i].getState() == Thread.State.NEW) {
      aHilos[i].setStartTime(startTime);
      aHilos[i].start();
    }
  }
}
void pararHilos() {
  for (int i=0; i<NHILOS; i++) {
    aHilos[i].pararHilo();
  }
}
void crearHilos() {
  String lines[] = loadStrings("BaseDatos.txt");
  for (int i=0; i<NHILOS; i++) {
    String words[] = lines[i].split(",");
    aHilos[i] = new Hilo(kinect, Integer.parseInt(words[0]),
tiempo, pipe);
    aHilos[i].setName(words[2]);
  }
}
void insertarUsuario(int userID) {
  for (int i=0; i<NHILOS; i++) {
    aHilos[i].setUser(userID);
  }
}
void keyPressed() {
  if (key == 'r') {
    startTime = clock.startTime();
    loop();
  }
```

```
}
  if (key == 'f') {
    noLoop();
  }
  if (key == ESC) {
    servidor.pararHilo();
    pararHilos();
  }
}

void onNewUser(int userId) {
  println("start pose detection");
  kinect.startPoseDetection("Psi", userId);
}
void onEndCalibration(int userId, boolean successful) {
  if (successful) {
    println(" User calibrated !!!");
    kinect.startTrackingSkeleton(userId);
    startTime = clock.startTime();
  }
  else {
    println(" Failed to calibrate user !!!");
    kinect.startPoseDetection("Psi", userId);
  }
}
void onStartPose(String pose, int userId) {
  println("Started pose for user");
  kinect.stopPoseDetection(userId);
  kinect.requestCalibrationSkeleton(userId, true);
}
```